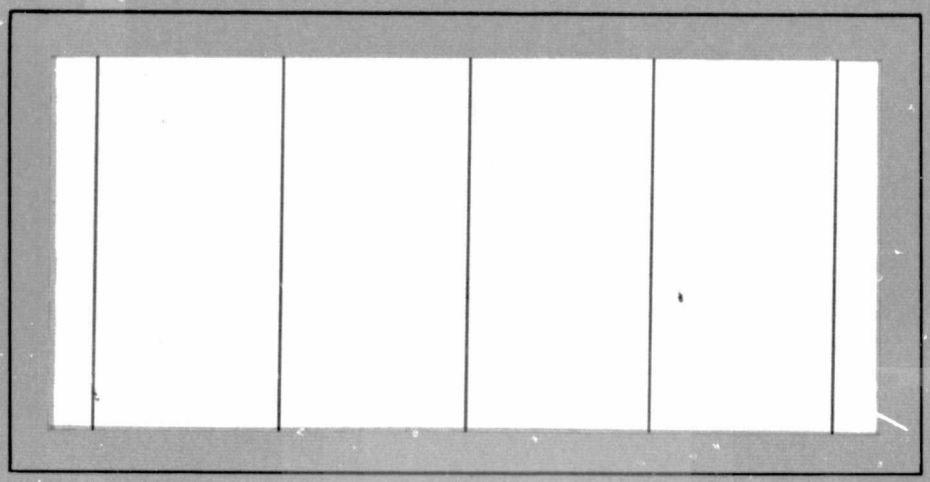


SCIENCE APPLICATIONS INCORPORATED

SAI-77-555-HU

NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM DESIGN FINAL REPORT PART II

SOFTWARE VERIFICATION/VALIDATION TECHNIQUES

CONTRACT NAS8-31379

Prepared under the direction of:

Mr. Bobby C. Hodges
Marshall Space Flight Center
National Aeronautics and Space Administration

March 19, 1976

S C I E N C E A P P L I C A T I O N S, I N C.

2109 W. Clinton Avenue, Suite 800, Huntsville, AL 35805
(205) 533-5900

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Initial work performed under this contract (in response to SOW Tasks Phase A, B, and C (Item 1) consisted of conducting a survey and analysis of the existing methods, tools, and techniques that were being currently employed in the development of software. As a result of this effort, we made the following recommendations for the construction of reliable software which pertain to all phases of the software development cycle:

- The software requirements stage should result in a structured, formal document which leads naturally into the software specification stage. It should be produced by an experienced analyst working in conjunction with the user. Origination of key software testcases should be an integral part of this stage.
- Software functional design specifications should be carried out through a formal language which is capable of reflecting fidelity of design with software requirements.
- Program code should be implemented using a structured programming language in which control structures are operationally apparent and as few in number as tolerable. Hence a structured preprocessor should be employed for code implementation if a structured compiler language isn't available.
- A programming language which promotes standardization of methods for accessing and operating on stored data bases such as the CODASYL Data Manipulation Language should be adopted and employed for purposes of data base verification.

- Software testing should be automated to establish user confidence while minimizing cost. Both static and dynamic testing are required. A static analyzer should enforce programming standards, while a dynamic analyzer should check the reliability of the code during execution. Structural and requirements testcase generators would greatly enhance the utility of the analyzers. A structural testcase generator produces data to test as many branches of the code as possible and should be employed for determination of software reliability. A requirements testcase generator produces data to determine the consistency of the code with the software requirements.
- Maintenance documentation needs to be an integral part of software. Documentation guidelines need to be established.

As work progressed, we translated our analysis results into viable software designs for three of the SSES tools. This work was done in accordance with SOW Tasks Phase A, items 1, 3, 4a, 4b, 4e, and 4f and Phase C, items 1 and 2. In the remainder of this report, we present functional software designs for the Software Specifications Language (SSL) (including a language reference manual) and the Data Base Verifier Subsystem (DBVS). In addition, a detailed (build-to) software design is described for the new capabilities to be incorporated into the static analyzer, FACES (FORTRAN Automated Code Evaluation System).

2. SOFTWARE SPECIFICATION

A primary goal of the SSES system is to provide early software feasibility and testing. Commensurate with this goal we have defined the Software Specification Language (SSL) for which the design is contained within this section. SSL is capable of representing all the information inherent in a functional block diagram of a software system. In addition, it is capable of (a) explicitly denoting the internal structure of data, (b) more completely denoting module interdependencies, and (c) expressing the input and output variables of each module. However, the most distinctive aspect of the language is the ability to attach requirement attributes to modules and variables which may be used in performing consistency checks.

The experience gained thus far in using SSL indicates that it requires slightly more effort than is normally applied to functional design. However, the effort is rewarded during detailed design as the module interconnections are much more evident and module functions tend to be more uniform and tractable.

2.1 THE LANGUAGE

2.1.1 Introduction

SSL (Software Specification Language) is a new formalism for the definition of specifications for software systems. The language provides a linear format for the representation of the information normally displayed in a two-dimensional module inter-dependency diagram. In comparing SSL to FORTRAN or ALGOL, one finds the comparison to be largely complementary to the algorithmic (procedural) languages. SSL is capable of representing explicitly module interconnections and global data flow, information which is deeply imbedded in the algorithmic languages. On the other hand, SSL is not designed to depict the control flow within modules. We refer to the SSL level of software design which explicitly depicts inter-module data flow as a functional specification.

We wish to express our appreciation to Mr. Bobby Hodges of Data System Labortory, George C. Marshall Space Flight Center for his guidance and support in the performance of this task.

2.1.1.1 Need for SSL

The current state of the art in software development permits insufficient formal evaluation prior to implementation. Such questions as:

- Are all requirements fulfilled?
- Have all software elements been defined?
- Are the element interconnections consistent?

cannot be answered in a manner that is independent of the designer's opinion. The intent of SSL is to formalize, through a language, the statement of the functional specification for a software system. Given this formal statement expressed in SSL and a translator for the SSL language, an independent evaluation of the software may begin much earlier in the development cycle.

In addition to evaluation, other aspects of SSL can aid both the designer and implementer. Several things that are characteristically omitted or inadequately performed during early design but required in SSL are:

- A complete and consistent statement of the software requirement
- Unambiguous communication of software organization to the detailed designer
- Enumeration of intraprogram consistency checks (assertions) useful during checkout.

A translator also provides tables and summaries for the final software documentation and a software element cross reference file. The latter could be used to statically verify the fidelity of the final code to original specifications.

2.1.1.2 Unique Features of SSL

The major contribution of SSL is the formal approach it brings to a phase of software development previously relegated to heuristic techniques as discussed above. Within this framework, there are several unique technical features possessed by SSL. First, the projection of a specialized form of software requirements onto the objects being defined establishes a rationale for the software structure not present in other methodologies. These requirements are an important aspect of consistency checking when evaluating a specific functional design. Second, the incorporation of levels of abstractions directly in a design methodology is a step forward in software engineering. Lastly, an automated SSL translator is being designed that is one of several interlocking software design and evaluation tools collectively called Software Specification and Evaluation System (SSES). SSES includes a static code analyzer, a dynamic code analyzer, and a test case analyzer. The specific capability that SSL brings SSES is the ability to test and evaluate software design early in the development cycle.

SSL also incorporates a flexible data abstraction capability and places emphasis on assertions as a means of describing the dynamic behavior of the software being designed. Although neither of these is unique, they are relatively new concepts in the field of computer science.

2.1.1.3 Background

In evaluating a new software system, particularly a programming language, it is important to trace the historical developments to which it relates and upon which it is based. The MIL (Module Interconnection Language) system [1] was a principal contributor to the concepts of data creation and data availability restrictions among modules within SSL. Guidelines imposed for the partition of programs into sub-systems are derived from the principles embodied in the concept of levels of abstraction [2]. Module descriptions in SSL are a linearlized form of the information available in the two-dimensional diagrams referred to as structure charts[3]. The data description capability is largely the same as that of PASCAL [4]. The syntax for expressions is derived from, but not identical to, that of ALGOL 60 [5]. Assertions in SSL have the form and appearance of those in the language NUCLEUS [6].

2.1.2 The Grammar

The material in this section is arranged in the form of a reference guide to the language, and not tutorially in the manner of a user's manual. To aid the reader, a cross reference index is provided in the last section.

2.1.2.1 Metalanguage Description

For the purposes of automatic translation and unambiguous communication, it is desirable to express SSL via a formal grammar. The vehicle selected for this purpose is the Backus-Naur-Form (BNF) metalanguage [5]. BNF has the advantages of being well known and compact in representation. In addition, most formal methodologies for analyzing grammars are based upon BNF representation.

Any nontrivial language contains an infinite number of legal sentences. Each sentence, in turn, is composed of the concatenation of strings; strings are composed of characters. A grammar uses strings as operands and combines them under the operation of concatenation to finitely depict, all legal sentences. The way in which this is done in BNF can best be interpreted via an example. Consider the following production:

<ab> :: = a|b| <ab> a

Sequences of characters enclosed within the brackets < >represent metalinquistic variables called nonterminal symbols. The marks "::=" and "|" are metalinquistic connectives meaning "is composed of" and "or" respectively. Any string not a nonterminal or connective denotes itself and is called a terminal symbol. Juxtaposition of symbols between connectives in a formula, such as the example, signifies that the symbols must be in the exact order denoted. The above production indicates that <ab> may have the values:

- a
- b
- a, aa, aaa, ...
- b, ba, baa, ...

In BNF, the null string is designated by <empty> :: =

SSL is represented as a context-free grammar which means:

- There exist a finite number of productions of the type of the above example.
- The left part of each production (i.e., left of ::=) consists of a single nonterminal symbol.
- There exists a unique nonterminal symbol (called the distinguished symbol) which is in the right part of no production except its own.

2.1.2.2 Overview of SSL Grammar

Prior to examining the detailed structure of SSL components, it will be useful to identify the overall structure of a software specification expressed in SSL. Figure 2-1 depicts the sequencing of the syntactical items used to describe an SSL specification.

A specification consists of one or more subsystems, each but the first having a name. The first subsystem is referred to as the "main" subsystem and each subsystem is composed of a preamble and one or more module descriptions. The preamble defines the local environment for the subsystem

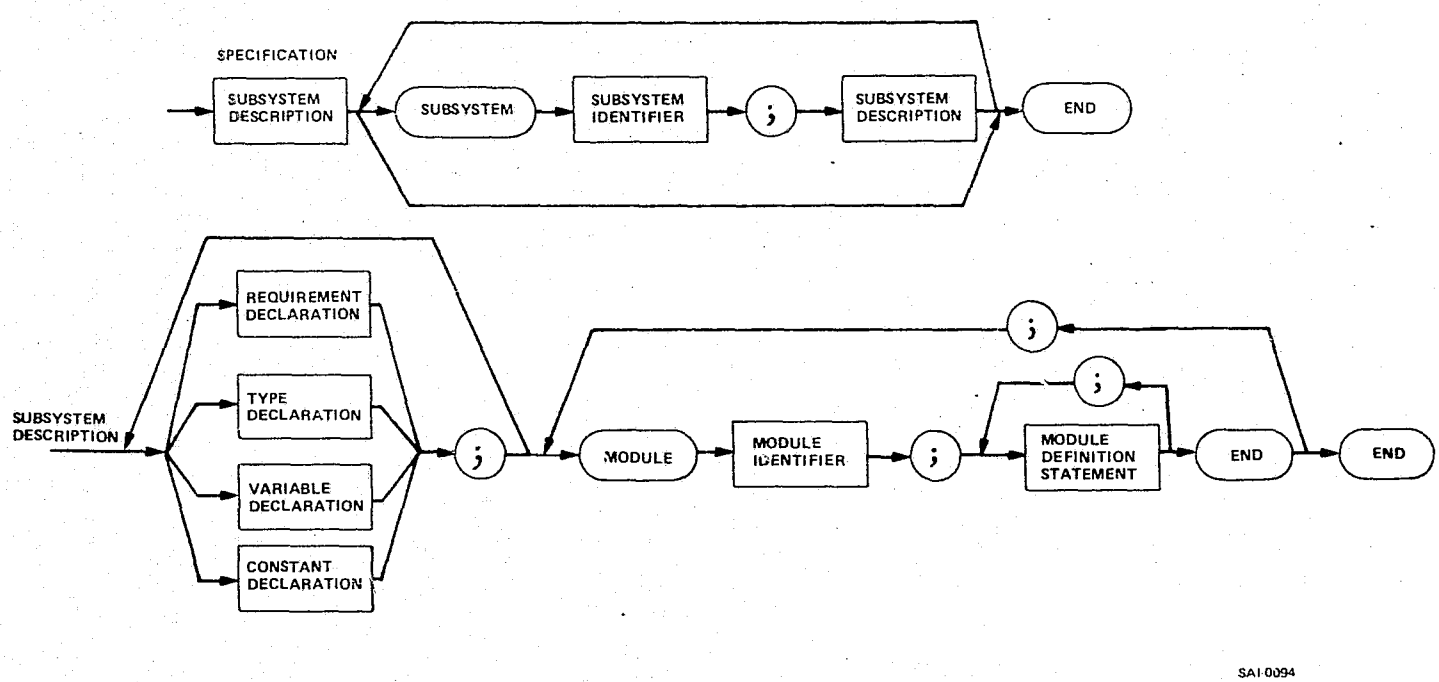


Figure 2-1 Syntactical Overview of an SSL Specification

(constants, requirements, data formats, etc.) and the module descriptions indicate operational aspects of program units (program units are subprograms, procedures, etc.).

In the following subsections, the detailed syntactical descriptions will be presented.

2.1.2.3 Basic Vocabulary

The basic vocabulary of SSL consists of special symbols, letters, digits, and reserved words. Each special symbol (Table 2-1) is primarily a single character except where limited computer character fonts require the concatenation of two characters. Where a special symbol consists of more than one character, it must be written without an intervening blank. Subsequently, special symbols other than ".", "@", and "_" will be referred to as delimeters. Each character in Table 2-1 is available within the ANSI standard codes [7] for ASCII-8, EBCDIC-8, and HOLLERITH-256. Substitutions may be necessary if an SSL translator is implemented in an environment not conforming to the standard character codes.

Letters and digits do not have individual meanings but are used to construct identifiers, numbers, and reserved words. The following basic productions enumerate these elements of the vocabulary:

<letter> ::= a|b| ... |z

<digit> ::= 0|1| ... |9

TABLE 2-1 SSL SPECIAL SYMBOLS

+	[
-	]
*	..
/	**
=	<=
,	>=
;	¬=
:	/*
<	*/
>	.
(	@
)	_
¬	

Reserved words (Table 2-2) are composed entirely of sequences of letters. In this document, they are normally underlined. A reserved word may not contain imbedded blanks and must always be followed by a blank or a delimeter.

The construct

```
/*  any sequence of symbols not containing "*/" */
```

may be inserted between any two identifiers, numbers, delimeters, or reserved words. It is called a comment any may be removed from the program text without altering its meaning.

2.1.2.4 Basic Language Elements

2.1.2.4.1 Identifiers

Syntax

```
<identifier> ::= <letter>|<identifier> <letter>
                 |<identifier> <digit>|<identifier> _
```

Examples

Legal	Illegal
a	5ad
b27	sr$p
cr_14dr	

TABLE 2-2 SSL RESERVED WORDS

Access	Fulfil	Transduction
Accesses	Fulfills	Transductions
Analog	Global	Transmit
And	Implies	Transmits
Array	In	True
Assume	Input	Type
Assumes	Inputs	Types
Boolean	Integer	Use
Case	Iteratively	Uses
Char	Modify	Using
Conditionally	Modifies	Variable
Constant	Module	Variables
Constants	Of	
Constraint	Or	
Constraints	Output	
Create	Outputs	
Creates	Real	
Digital	Receive	
Doubleprecision	Receives	
End	Record	
Entry	Requirement	
Equ	Requirements	
Execute	Satisfies	
Executes	Satisfy	
False	Set	
File	Subjectto	
For	Subsystem	
Forall	Text	
From	To	

Semantics

Identifiers must begin with an alphabetic character and contain only letters, digits, and the "_" symbol. The latter is known as the break character. Identifiers have no inherent meaning, but serve as identification for variables, modules, subsystems, and other elements of a software specification.

Identifiers may be of arbitrary length but must be unique within the first twelve characters. No identifier may be equivalent to the first twelve characters of a reserved word. The same identifier may not be used to denote two different quantities within a subsystem with the exception of field names in different records.

2.1.2.4.2 Numbers

Syntax

```
< unsigned integer>  ::=<digit> | <unsigned integer>
                          <digit>

< sign   > :: =  + |  -

< exponent  part > :: =  e <unsigned integer >
                        |e <sign> <unsigned integer>
                        |d <unsigned integer>
                        |d <sign> <unsigned integer>
<  decimal number > :: = <unsigned integer>
                        |<unsigned integer> .
                         <unsigned integer>
<  unsigned number >:: = <decimal number >
                        |<decimal number> <exponent
                         part>
```

Examples

Legal	Illegal
57	3,746
14d10	XII
3.7 e-5	e+7
0.2	.14

Semantics

Decimal numbers have their conventional arithmetic meaning. The exponent is a scale factor expressed as an integal power of 10. A number expressed with neither a scale factor nor a decimal fraction is assumed to be of type integer. A number which uses the "d" form for the exponent part is assumed to be double precision. Otherwise, the number is assumed to be type real. Note that if a number contains a decimal point, at least one digit must precede and succeed the point.

2.1.2.4.3 Logical Values

Syntax

<logical value> ::= true | false

Semantics

Logical values have their conventional meaning and may be defined by describing their combination under the operations "union" and "intersection". The union of the logical value true with any other logical value always yields the result true. The intersection of the logical value false with any other logical value always yields the result false.

2.1.2.5 Requirement Declaration

The several parts of the requirement declaration are used to identify the data flow between the software package being described and other parts of the total system. In addition, they identify processing steps (called transductions) and restrictions (called constraints) which are attached to both modules and variables.

Syntax

<requirement declaration> ::= <requirement or requirements>
<requirement statement group> end

<requirement or requirements> ::= requirement
| requirements

<requirement statement group> ::= <requirement statement part>
| <requirement statement group> ;
<requirement statement part>

<requirement statement part> ::= <input part>
| <output part> | <transduction part>
| <constraint part>

2.1.2.5.1 Input and Output Parts

An input is a system level input (or stimulus) which a software package receives from an external source. An output is a system level response which has a purpose beyond the immediate concern of the software package being described.

Syntax

<input part> :: = <input or inputs> <entire variable list>

<output part> :: = <output or outputs> <entire variable list>

<input or inputs> :: = input|inputs

<output or outputs>:: = output|outputs

Examples

- input state_vector ;
- inputs mass, velocity, distance ;
- output concordance_list ;

Semantics

A variable may be in both an input and an output list. A variable in an output list not used within the subsystem other than in the module in which it is initialized is not required to have a requirement transduction attribute. The structure of all variables in input and output lists must be described within the variable statements of the subsystem preamble. Each subsystem preamble must have a requirement declaration with an output part.

2.1.2.5.2 Transduction Parts

Transductions are identifiers representing processing steps. They are derived by first writing a high level pseudo-program to "transduce" the input variables into the output variables and then extracting and listing the major verbs of the program. Just as the processing steps of the pseudo-program may be nested, the transductions may likewise be nested. Ideally, for each subsystem there should be from three to seven transductions that are not nested within any others.

Syntax

< transduction part> :: = <transduction or transductions >

<transduction clause>
|<transduction part> ; <transduction clause >

< transduction or transductions >:: = transduction
| transductions

< transduction clause> :: = <transduction list>
|<transduction list> in <transduction list >

< transduction list> :: = <transduction identifier>
|<transduction list >, <transduction identifier>

< transduction identifier >:: = <identifier>

Examples

- transduction sum_expense, sub_deduct in tax_compute; write_paycheck;

- transductions save_options; read_card in parse;

Semantics

Within a transduction clause, each processing step represented by a transduction identifier to the left of in must be a substep of the processing steps listed on the right of in. Each transduction identifier represents a unique processing step, but may be reused to show different substep relationships. Substep relationships must be consistent, i.e., the complete set of substep relationships partially order the transduction identifiers.

2.1.2.5.3 Constraint Parts

Syntax

< constraint part> ::= <constraint or constraints>
< constraint list>

< constraint or constraints >::= constraint
| constraints

< constraint list > ::= <constraint identifier >
|<constraint list >, <constraint identifier>

< constraint identifier> ::= <identifier>

Examples

- constraint carpool_size ;
- constraints max_targets, minimum_distance ;

Semantics

Each constraint identifier defined must be attached as an attribute to some module in the subsystem.

2.1.2.6 Data Type and Variable Declarations

Explicit description of data and the ability to define and use new data types is one of the greatest assets of SSL. A new data type may be described directly as part of a variable declaration, or described independently for subsequent use.

Syntax

< type declaration> ::= <type or types>
<type definition>
|< type declaration> ; <type definition>

< type or types> ::= type | types| global type
|global types

<type definition> ::= <identifier> = <type>

<type >::= <simple type> |<structured type>

|<pointer type>

<variable declaration> ::= <variable or variables>
<variable definition>
| <variable declaration> ; <variable definition>

<variable or variables> ::= variable | variables

<variable definition> ::= <identifier list> :< type>

|<identifier list> : <type> ; <for clause>
|<identifier list> : <type> ; <subjectto clause>
|<identifier list> : <type> ; <for clause>;
<subjectto clause>

<for clause> ::= for <transduction list>
<subjectto clause> ::= subjectto <assertion list>
<assertion list> ::= <assertion>

|<assertion list> ; <assertion>

<identifier list> ::= <identifier> |<identifier list> ,
<identifier>

Semantics

A type declaration list is used to define new data types. Each type is named and may be referenced by the identifier to the left of "=" in the <type definition> production. The normal scope of a type identifier is the subsystem in which it is defined. However, the scope of a global type is the entire SSL program. Global types may be defined only in the main subsystem.

A data type need not be named if it is defined intrinsic to the variable declaration. Both type and variable declarations may use data types defined and named elsewhere. Examples of both are given in the following subsections.

The <for clause> of the variable declaration is used to attach requirement attributes. Requirement attributes limit the availability of variables within the modules of the subsystem. All variable declarations must contain a for clause with the exception of output variables identified in the requirement statement.

The <subjectto clause> identifies the global assertions associated with the variables being declared. A global assertion is one that must be true upon exit from the module creating the variable, and true on both entry and exit of modules using the variable.

2.1.2.6.1 Simple Types

Simple types are data types for which the designer, using SSL, need not define the internal structure or the internal structure has previously been defined and named.

Syntax

<simple type> ::= <basic type> |<scalar type>
|<subrange type> | <type identifier>

<type identifier> ::= <identifier>

Semantics

A type identifier must previously have been used to the left of an "=" in a type definition.

2.1.2.6.1.1 Basic Types

The basic data types are those which are implicity defined by the SSL language.

< basic type> ::= integer| real| boolean
|doubleprecision |char| analog| text

Examples

- variables I, J, K: integer; for count_people ;

- variables height: real;
  for record_status;
  subjectto height >0.0 ;
  height <= 10.0 ;

  employed: boolean ;
  for record_status ;

Semantics

The types integer, real, boolean, and doubleprecision have the conventional meaning. The type char indicates a single unit of hollerith information. Type text indicates hollerith data with unspecified length. Since the length of a text item varies, it may not be combined with other variables in forming structured data types. The type analog designates a data item which contains analog signal information. Like the text type, it may not be combined with other variables to form structured types.

2.1.2.6.1.2 Scalar Types

Scalar types are used to designate a finite number of disjoint states which a variable may represent. In conventional programming languages, it is customary to declare the variable of type integer and assign it only the cardinal numbers 1, 2,..., n where each value represents one state of the several possible.

Syntax

<scalar type> ::= (<identifier list>)

Examples

- type marital_status = (single, married, divorced);
  variable ms: marital_status; for emp_record;
- variable color: (Red, blue, yellow, green) ;

Semantics

Conceptually, the elements of scalar types are ordered regardless of whether or not the underlying set of states is ordered. The order is always the same as that of the identifiers in the identifier list. This enables a designer to use relational tests (< ,>, etc.) in assertions involving scalar type variables.

2.1.2.6.1.3 Subrange Types

Subrange types are used to designate a subset of integers or scalars which a data item may assume.

Syntax

<subrange type> ::= <constant> .. <constant>
<constant> ::= <unsigned integer>
| <sign> <unsigned integer>
| <constant identifier>

|<sign> <constant identifier>

|<logical value>

< constant identifier> ::= <identifier>

Examples

- variables weight: 10..350; for ins_compute;
  dependents: 0..15; for tax_compute;

- type color = (purple, blue, red, yellow, green, black);
  variable primary_color: blue..green;

Semantics

A subrange simply indicates the least and largest constant values an item may assume. The lower bound (left-most constant in the production) must be less than the upper bound. A subrange with bounds expressed in types other than integer or scalar is not permitted.

Constant identifiers may arise from two contexts. The first is the appearance of an identifier to the left of "=" in a constant declaration. The second (illustrated by the above example) is the appearance of an identifier as a scalar element. Constant identifiers arising from the second context may not be preceeded by a unary sign.

2.1.2.6.2 Structured Types

A structured type is a data type composed of more elementary data types.

Syntax

< structured type> ::= <array type>|<record type>
|<digital type>| <set type>
|<sequence type>

Semantics

An SSL structured data type is used to indicate the general form and content of a data structure, not precise implementation word and storage formats. In SSL, the following definitions are used:

Array - A fixed number of data items, all of the same type and length and accessed by computed index.

Record - A fixed number of data items, each of fixed length, and each equally accessible.

Digital- A record having additional restrictions which are discussed in a subsequent subsection.

Set - An element of the powerset of a finite number of basic elements.

Sequence or File - A variable number of data items, all of the same type and length; however, each element is not equally accessible at all times.

Stronger connotations (such as elements of an array are sequentially stored) are not implied by the semantics of SSL.

2.1.2.6.2.1 Arrays

An array is a fixed number of data elements, each of the same type and length and each equally accessible. Elements of an array are ordered and each element is accessed by a cardinal number called its index.

Syntax

< array type >::= array [<index list>] of
<component type>

< index list> ::= <index type> |<index list> ,
<index type>

< index type >::= <simple type>

< component type> ::= <type>

Examples

- variable matrix: array [1..10, 0..20] of real;
for ta ;

- type people = (adams, buckles, jones, smith);
variable employee: array [people] of 1..50 ;
for ta;

Semantics

Index types must have a finite range and be ordered. This requirement eliminates index type of integer, real, and double-precision. However, subranges of integers are permitted. For the purpose of ordering, false <true for boolean type indices.

2.1.2.6.2.2 Records

A record is a structure containing a number of components called fields. Fields are not constrained to be of identical type but must be of fixed length. A single record type is permitted to have variants.

Syntax

```
<record type> ::= record <field list> end
<field list> ::= <fixed part> | <fixed part> ;
        <variant part> |<variant part>
<fixed part> ::= <record section> |<fixed part> ;
        <record section >
<record section> ::= <field identifier list> : <type>
<field identifier list>  ::= <field identifier>
      |< field identifier list> , <field identifier>
<variant part> ::=  case  <tag field> <type identifier>
        of  <variant list>
<variant list> ::= <variant> | <variant list> ;
        <variant>
<tag field> ::=  <field identifier>   :
<field identifier> ::= <identifier>
<variant> ::= <case label list> :  (<field list> )
        |<case label list> :( )
<case label list> ::= <case label>  |<case label list> ,
         < case label>
<case label> ::= < constant>
```

Examples

- Type employee = Record

```
        Number:Integer;
        Salary:Real;
        Name:Array [1..24] of char
    End;
```

- Variable machine_part:Record

```
        Part_No, Order_Quantity:Integer;
        Weight:Real
    End;

    for customer_billing;
```

- Type complex = Record real_part, imag_part:real End;

- Type farm =(peaches, cotton, soybeans);

```
  Type  land_use = Record

        Owner_Name:Array [1..12] of char;
        Plot_No:Integer;
        Case Crop:Farm of

              peaches:(tree_count:Integer);
              cotton, soybeans:(plant_date:Integer ;
                  herbicide, insecticide:boolean)

    End;
```

- Variable sizes:Array [1..10] of Record

```
              Height:Integer;
              Weight:Real

        End;
```

```
for  health_file_update;
subjectto height >0; height <120;
          weight >0.0;  weight  <500.0 ;
```

Semantics

Fields may not be of basic types text or analog. A record may be a component of another record, but a digital type may not. The scope of a field identifier is the smallest record in which it is defined. Field identifiers with disjoint scopes may be reused. Access of a component is always by the field identifier and never by a computed value.

The type associated with the tag field of a variant must contain only a finite number of elements. This limits it to boolean, subrange, and scalar. All elements of the type must appear in some case label list of the variant. If the field list for case label L is empty, the form is:

L : ( )

A record may contain only one variant part and it must succeed the fixed part. However, a variant may contain variants. That is, it is possible to have nested variants. All field names of the same record must be unique even if they are in different variants.

2.1.2.6.2.3 Digital Types

Digital types are a restricted form of records to represent real time digital signals.

Syntax

<digital type> ::= digital <fixed part> end

Example

- Variable Signal_In: digital
  Valve_1: boolean;
  LOX_Switch: 1..3;
  Command: (Idle, stopped, running)
  End;
  for check_status;

Semantics

Due to their physical interpretation, the type of components within digital types may only be boolean, scalar, or subrange. Digital types may not have variant parts and they may not be used as components of any other type.

2.1.2.6.2.4 Set Types

Set types represent elements of powersets over a finite set of elements called the base type. Conceptually, a set type variable may be viewed as a bitstring of length equal to the number of elements in the base type. Each bit is associated with a unique element and is "on" or "off" if the element is a member or not a member of the powerset.

Syntax

<set type> ::= set of <base type>
<base type>::= <simple type>

Examples

- Type members = (father, mother, big_sister,
  little_sister, big_sister, little_brother);
  Variable family: set of members; for arrange;
- Variable Even_numbers: set of -10..10;
  for compute_something ;

Semantics

The base type must be either scalar or subrange.

2.1.2.6.2.5 Sequence (File) Types

A sequence differs from an array in that it may vary dynamically in length and is referenced through a "window" called its buffer (not by computed index). Examples of physical representations of sequences include linked lists and mass storage files.

Syntax

<sequence type> ::= <file or sequence> of <type>
<file or sequence> ::= file | sequence

Examples

```
• Variable Assembly: sequence of record
                part_name: array [1..6] of char;
                order_no: integer;
                drilled, punched, stamped, purchased:
                  boolean
          End; for update_orders ;

• Type roster_entry = record
                name: array [1..20] of char;
                rank: 1..16; base_code: 1000..5000
          End;
  Variable roster: file of roster_entry;
                for assign_new_base ;
```

Semantics

All components of sequences must be of identical type and length. A sequence may not have sequence type or text type components. Furthermore, digital and analog types may not be combined as sequences.

2.1.2.6.3 Pointer Types

Variables of type pointer are "bound" to a particular type. That is, the contents of a pointer is used to indicate a second variable, and the second variable is required to be of a predetermined, specific type.

Syntax

<pointer type> ::= @ <type identifier>

Examples

- Type combination = record n, p: integer End ;
  Variable comb_ptr: @ combination; for select_band;

- Type weather_station = record hi,lo: integer;
  rain: real End;
  Variable ws_ptr: @ weather_station;
  for record_temperature;

Semantics

The contents of a pointer may be altered, but the data element the pointer indicates is always of the same type.

2.1.2.7 Constant Declarations

In SSL, constant declarations may appear in the preamble of any subsystem and are used to communicate actual values or parameters to the detailed designer. Normally, a constant declaration would be used only for critical values for which the effects are to be isolated in the final code.

Syntax

<constant declaration> ::= <constant or constants>
<constant definition list>
<constant or constants> ::= constant | constants
<constant definition list> ::= <constant definition>
| <constant definition list>;<constant definition>
<constant definition> ::= <identifier> = <constant>
| <identifier> = <simple type>

Examples

- Constant a = 10.0 ; max_count = Integer;
- Constants Low = true;
Tax_cut = 1..5 ;

Semantics

An identifier declared equal to a simple type indicates that the exact value is not known at the time of specification, but will be provided before implementation. An identifier used in a constant declaration may subsequently be used any place that a constant (of the same type) may be used.

2.1.2.8 Data References

Data elements may be referenced by variable name, by selected component, or pointer. A variable has components only if it is a record, digital signal, file, or array.

Syntax

< variable> ::= <entire variable>
| <component variable>
| <referenced variable>

2.1.2.8.1 Entire Variables

< entire variable> ::= <identifier>

Semantics

A reference to an entire variable includes all fields of a record or digital signal, all elements of an array, or all records of a file. If the data element is a simple, unstructured variable (integer, boolean, etc.) it may only be referenced as an entire variable.

2.1.2.8.2 Component Variables

Syntax

< component variable> ::= <indexed variable >
| <field designator>
| <file buffer>

< indexed variable> ::= <array variable>
[<expression list>]

<array variable> ::= <variable>

<expression list> ::= <expression>|<expression list> ,
<expression>

<field designator>::= <record variable> . <field
identifier>

<record variable> ::= <variable>

<file buffer> ::= <file variable> @

<file variable> ::= <variable>

Examples

Char_Array [15]

Inverse_Matrix [5, I, 16]

Employee.Name

Owner [15] . Accessed_Value

Name_Record.Character [6]

Transaction_File @

Transaction_File @ . Date

Transaction_File @ . Date. Month

Semantics

Indexed variables have the conventional meaning. Field designators denote which field component of a record or digital signal type is to be selected. A file buffer variable designates the current active element of the sequence of elements that comprise the file.

Since arrays, files, and records can be combined in various ways (a record of records, file of arrays, array of records, etc.) a component variable can be arbitrarily complex. It is recommended that data structures be as limited in complexity as the problem permits.

2.1.2.8.3 Referenced Variables

Syntax

<referenced variable> ::= <pointer variable> @

<pointer variable> ::= <variable>

Examples

```
Symbol_Pointer @
Student_Name [6] @
Assembly@.Manufacturer@
```

Semantics

The data structure denoted by the contents of the pointer variable is substituted for the referenced variable in expression evaluation.

2.1.2.9 Expressions and Assertions

Expressions arise in two contexts: subscripts of arrays and as terms within assertions. Assertions may appear in either variable declarations or module descriptions.

2.1.2.9.1 Arithmetic Expressions

Arithmetic expressions in SSL are similar to those in other high level languages. Results of expressions are single valued with type determined by the operation and the constituent operands.

Syntax

<arithmetic expression> ::= <term> | <sign> <term> |< arithmetic expression> <sign> <term>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<factor> ::= <primary> | <factor> ** <primary>|<set>

<primary> ::= <constant identifier> | <unsigned number>| <variable> | < function designator > |(<arithmetic expression>)

<set> ::= [<element list>]

<element list> ::= <empty> | <element> | <element list>, <element>

<element> ::=<expression> | <expression> .. <expression>

< multiplying operator> ::= * |/

<function designator> ::= <function identifier> (<expression list>)

<function identifier> ::= <identifier>

Examples

```
a + b
3.0 * sin ( r + 1.0)
2 * (ifix(c) + blank_common.icount)
name.field1
name_set + [joe, fred]
```

Semantics

Mixed mode expressions are prohibited with the exception of the exponentiation operator as indicated in Table 2-3. In Table 2-3, any operand of type integer may be replaced by an operand of type integer subrange. The symbol "dp" indicates double precision. The unary "+" may be used with any operand permitting a unary "-", but is semantically superfluous (i.e. + is the identity operation). If a type is not included in the operand type columns of Table 2-3 then its use with the designated operator is not permitted. Note, however, that integer and integer subrange are interchangable.

SSL does not contain intrinsically defined functions. All function identifiers are accepted, but it is suggested that those embodied in the proposed implementation language be adopted for each specification. Function types are not explicity declared, but must be consistently used throughout the specification. In addition to the basic types (integer, real, etc.), the permissible function types include scalar and subranges of integers and scalars.

TABLE 2-3 ARITHMETIC OPERATIONS

Operator	Operation	V1 "OP" V2 V1 Type	V2 Type	Result Type
-	Arithmetic Negation		Integer	Integer
			Real	Real
			dp	dp
+,-	Addition, Subtraction	Integer	Integer	Integer
		Real	Real	Real
		dp	dp	dp
+,-	Set Union, Set Difference	Set	Set	Set
*,/	Multiplication, Division	Integer	Integer	Integer
		Real	Real	Real
		dp	dp	dp
*	Set Intersection	Set	Set	Set
**	Exponenciation	Integer	Integer	Integer
		Real	Integer	Real
		Real	Real	Real
		dp	Integer	dp
		dp	Real	dp
		dp	dp	dp

2.1.2.9.2 Boolean Expressions

Combining arithmetic expressions with the boolean operations produces the expressions used in SSL assertions and array subscript lists.

Syntax

<expression> ::= <implication>|<expression> equ <implication>

<implication> ::= <boolean term>|<implication> implies <boolean term>

<boolean term> ::= <boolean factor>|<boolean term> or <boolean factor>

<boolean factor> ::= <boolean secondary>|<boolean factor> and <boolean secondary>

<boolean secondary> ::= <boolean primary>|¬<boolean primary>

<boolean primary> ::= <logical value>|<arithmetic expression>|<relation>|(<assertion>)

<relation> ::= <arithmetic expression><relational operator><arithmetic expression>

<relational operator> ::= <|<=| =| >= |¬ = | in

Examples

```
Rate = 7.0
Value and  Qual
a>b Implies c>0.0
S ¬= t  Equ  p<t
Color in [red, green, yellow]
abs (buffer @.velocity) <16.0 and  weight >= 14.0
```

Semantics

The arithmetic and boolean operators are grouped into hierarchial levels as exhibited in Table 2-4. Operations are performed in the order of highest hierarchial level first followed by equal hierarchial levels from left to right. This sequence may be overridden by parentheses, in which case the innermost operations are performed first. The meaning of the logical operators ¬ (not), and, or, implies, and equ (equivalent) is given in Table 2-5.

Table 2-6 depicts the required operand types for the boolean and relational operators. For set types, the symbols "[]" stand for the empty set. When comparing set types to scalars, the base type of the set must be the same as that of the scalars. The operators <, <=, =, >=, >, ¬ = stand for less than, less than or equal, equal, greater than or equal, greater than, and not equal respectively. Relational operators (other than in) may be used to compare arrays of equal length composed of characters, in which case they denote alphabetical ordering.

TABLE 2-4 OPERATION HIERARCHY

Level	Operations
1	Equ
2	Implies
3	Or
4	And
5	¬
6	<, < =, =, >=,> ,¬ =, In
7	+, -
8	*, /
9	**

TABLE 2-5 LOGICAL OPERATOR TRUTH TABLE

b1	false	false	true	true
b2	false	true	false	true
¬ b1	true	true	false	false
b1 And b2	false	false	false	true
b1 Or b2	false	true	true	true
b1 Implies b2	true	true	false	true
b1 Equ b2	true	false	false	true

TABLE 2-6 BOOLEAN AND RELATIONAL OPERATIONS

V1 "OP" V2

Operator	Operation	V1 Type	V2 Type	Result Type
<, <, =, >=, >, ¬ =	Compare	Integer	Integer	Boolean
		Real	Real	
		dp	dp	
		Boolean	Boolean	
		Char	Char	
		Scalar	Scalar	
In	Set Inclusion	Scalar	Set	Boolean
		Set	Scalar	
		Set	Set	
		Subrange	Set	
		Set	Subrange	
¬	Logical Inversion	Boolean	Boolean	Boolean
And	Logical "And"	Boolean	Boolean	Boolean
Or	Logical "Or"	Boolean	Boolean	Boolean
Implies	Logical Implication	Boolean	Boolean	Boolean
Equ	Logical Equivalence	Boolean	Boolean	Boolean

2.1.2.9.3 Assertions

Assertions are conditions which may assume only true/false values. They are attached to variables at their point of declaration and to modules. Module assertions depict entry and exit data conditions.

Syntax

<assertion> ::= <expression><forall clause>
<forall clause> ::= <empty>| forall identifier = <set>

Examples

- a [i] = 0.0 forall i = [1..n-1]
  (b.c [j] = t [k] forall j = [1,3,4..16]) forall k = [16..30]
- big>small
- code = 1 implies (eof equ true)

Semantics

The scope of the identifier in the <forall clause> is the assertion in which it is used and must not overlap that of a local or global variable of the same name. Its type is assumed to be the base type of the set within the <forall clause>. The set must represent a finite number of elements and may not be empty.

The expression within the assertion may assume only the values true and false. If the <forall clause> is present, the expression is evaluated once for each unique value which the <forall identifier> can assume from the set.

2.1.2.10 Module Descriptions

Modules are basic system objects in an SSL system description. In using SSL, one identifies for each module:

- The module name
- Input and output data
- Conditions placed on data upon entry to and exit from the module
- Dependence of the module on environmental objects and other modules

The rule of correspondence between input and output data is not stated in SSL. Its statement is a function of detailed design.

Syntax

<module description> ::= <module statement>;
<module definition part> end

<module definition part> ::= <module definition statement>|<module definition part> ;
<module definition statement>

<module definition statement> ::= <assumes statement>
|<satisfies statement>|<fulfills statement>
|<accesses statement>|<modifies statement>
|<creates statement>|<uses statement>
|<receives statement>|<transmits statement>
|<executes statement>

2.1.2.10.1 Module Statement

The module statement is always the first statement of a module description. It identifies the module by name and declares the local variables (if any).

Syntax

<module statement> ::= <module or entry> <module identifier> <release variable group>

<module or entry> ::= module|entry

<release variable group> ::= <empty>|(<release variable list>)

<release variable list> ::= <release variable>|<release variable list>; <release variable>

<release variable> ::= <variable >|<local variables>

<local variables> ::= <identifier list>:<simple type>

<module identifier> ::=<identifier>

Examples

- module matrix_multiply;
- entry push_stack (stack_item:stack_entry);
- module permutation (m, n:integer; elements:p_array);

Semantics

A module statement introduced by module can only be referenced from within the subsystem in which it is declared.

A module statement introduced by entry can be referenced only from subsystems other than the one in which it is declared.

Release variables occur both in module statements and virtual references within execute statements. Local variables within a release group serve strictly for communication between the module and those calling it. In this respect, they differ from global variables declared in the subsystem preamble which serve to communicate among modules having common requirement attributes. Local variable identifiers must be unique throughout a subsystem. Only the module statements introducing entry modules are permitted release variables which are not local variables. The variables of a release group for a module statement of an entry module must agree in type, number, and sequence to each virtual reference to it from other subsystems.

2.1.2.10.2 Assumes and Satisfies Statements

The assumes and satisfies statements specify truth conditions for data.

Syntax

<assumes statement> ::= <assume or assumes>
    <assertion list>
<satisfies statement> ::= <satisfy or satisfies>
    <assertion list>
<assume or assumes> ::= assume | assumes
<satisfy or satisfies> ::= satisfy | satisfies

Examples

- Assume a >0.0 ;
- Satisfies big_sister in family; count ¬┐ = 0 ;

Semantics

The assumes statement specifies data conditions that must be true upon module entry. The satisfies statement specifies data conditions that must be true upon module exit. Variables used in assertions must be either local variables in the release set or in the availability set pertinent to the module. (The availability set consists of those variables having requirement attributes which subsume all requirement attributes of the module.)

2.1.2.10.3 Fulfills Statement

The fulfills statement attaches requirement attributes to a module.

Syntax

<fulfills statement> ::= <fulfil or fulfills>
<requirement attribute list>

<requirement attribute list> ::= <attribute identifier>
|<requirement attribute list> , <attribute identifier>
<attribute identifier> ::= <transduction identifier>
|<constraint identifier>
<fulfil or fulfills> ::= fulfil | fulfills

Examples

- fulfills size_constraint, cluster;
- fulfil name_list ;

Semantics

All modules must have at least one transduction identifier attached as a requirement attribute. All attribute identifiers must be declared in the preamble to the subsystem in which the module is declared.

2.1.2.10.4 Accesses Statement

The accesses statement is used to indicate which environmental objects (chiefly peripherals) are utilized by a module.

Syntax

<accesses statement> ::= <access or accesses>
<environmental object list>

<access or accesses> ::= access | accesses

<environmental object list> ::= <environmental object identifier> | <environmental object list> , <environmental object identifier>

<environmental object identifier> ::=<identifier>

Examples

- Access line_printer;

- Accesses real_time_clock, system_disk ;

Semantics

For each environmental object there must be a unique identifier for which the scope is the entire specification.

2.1.2.10.5 Receives and Transmits Statements

The receives and transmits statements are used to indicate real time data activity such as is associated with telecommunications, analog, and digital signals.

Syntax

<receives statement> ::= <receive or receives>
<from clause>|<receives statement> ; <from clause>

<from clause> ::= <entire variable list> from
<environmental object identifier>

<transmits statement> ::= <transmit or transmits>
<to clause>|<transmits statement> ;
<to clause>

<to clause> ::= <entire variable list> to
<environmental object identifier>

<receive or receives> ::= receive | receives

<transmit or transmits> ::= transmit | transmits

<entire variable list> ::= <entire variable>
|<entire variable list> , <entire variable>

Examples

- Receive weight from strain_gage_1;

- Transmits course_correction to ground_control;

Semantics

The scope of the environmental object name is the entire specification. Note that components of structured variables may not be transmitted or received.

2.1.2.10.6 Creates, Modifies, and Uses Statements

The creates, modifies, and uses statements distinguish between input and output data variables. They may also indicate how the two are related in a manner short of a rule of correspondence. A complete rule of correspondence (algorithm) is a task of detailed design and not of SSL.

Syntax

<creates statement> ::= <create or creates>
<create list>

<modifies statement> ::= <modify or modifies>
<modify list>

<modify list> ::= <variable list><using clause>
|<modify list>; <variable list><using clause>

<create list>::= <entire variable list><using clause>
|<create list>;<entire variable list><using clause>

<uses statement> ::= <use or uses> <variable list>

<create or creates> ::= create|creates

<modify or modifies> ::= modify|modifies

<use or uses> ::= use|uses

<using clause> ::= <empty>|using <variable list>

<variable list> ::= <variable>| <variable list >,
<variable>

Examples

- create employee_array using name_file;
- modifies count, fica_rate using tax_table,
salary_scales;
- modify pressure.weight [4] , names [10] .initials;
- uses cluster@, transaction_file;

Semantics

The order of the variable references in any variable list has no significance.

The variables within a using clause or a uses statement are input variables. A variable may be both input and output. An input variable in a using clause indicates that its contents are instrumental in determining the final contents of the output variables within the same statement extending to the first semicolon on the left.

The presence of a variable in the output list of a creates statement indicates the first use (in a dynamic sense) of that variable. This does not mean, however, that the variable may not appear previously in the sequential listing of the SSL program. The implication of the creates statement is that all variables in the output list are first computed or initialized in the module being described. All variables declared in the subsystem preamble must appear as an output variable in exactly one creates statement within the subsystem unless it is a release variable of an entry module.

All variables appearing in a creates, modifies or uses statement (other than the output list of the creates statement) must be in the availability set for the module. A variable is in the availability set of a module if the transduction requirement attributes of the variable subsume all the transduction requirement attributes of the module.

2.1.2.10.7 Execute Statement

The execute statement designates modules which are called by the module being described. It may indicate that specific modules are called iteratively, conditionally, or both.

Syntax

<executes statement> ::= <execute or executes>
<call list>|<executes statement>; <call list>

<call list> ::= <module reference list>| <module reference list> <call list tail>
|<call list tail>

<call list tail> ::= <iteratively clause >
|<conditionally clause>

<iteratively clause> ::= iteratively <module reference list> | iteratively <call list tail>

<conditionally clause> ::= conditionally <module reference list>

<execute or executes >::= execute|executes

<module reference list> ::= <module reference>
|<module reference list> , <module reference>

<module reference> ::= <concrete reference>
|<virtual reference>

<concrete reference> ::= <module identifier>

<virtual reference> ::= <subsystem identifier> .
<module identifier><release variable group>

Examples

- Execute matrix_multiply, cluster·group (pointer@);
- Execute iteratively suba, subb;
  conditionally subc, subd, sube;
- Executes sqrt; iteratively cos conditionally sin;

Semantics

The order of module identifiers in the module reference lists is not significant. The domain of either an iteratively or conditionally clause extends to the next semicolon. An iteratively clause may overlap another clause.

Presence of a module identifier in a iteratively clause connotates that it is called from within a loop. Presence in a conditionally clause connotates the module is not always called. If present in neither, the module is called unconditionally but not from within a loop.

A concrete reference is a call to a module within the same subsystem. A concrete reference may never be to an entry module. A virtual reference is a call to a module of a different subsystem and must always be to an entry module.

Within the release variable group, the local variable format must be used for variables never before defined. A variable may have been defined in the preamble to the subsystem or in the last module statement. The entry module to which the virtual reference refers must have the same release list with respect to number, order, and type of variables. All variable types used in a virtual reference release list must be either intrinsically defined (boolean, real, text, etc.) or global types.

2.1.2.11 Subsystem Descriptions

Subsystems are independent software units, each with its own requirement declaration. Subsystems may not share global variables but communicate via the release group variables of virtual references and entry modules. The only identifiers with scope greater than a single subsystem are global type identifiers, environment object identifiers, subsystem identifiers, and function identifiers.

Syntax

<subsystem description> ::= <subsystem preamble> ;
<module description list> end

<module description list> ::= <module description>
|< module description list>; <module description>

<subsystem preamble> ::= <preamble declaration list>
| subsystem <subsystem identifier> ; <preamble declaration list>

<subsystem identifier> ::= <identifier>

<preamble declaration list> ::= <preamble declaration>
|<preamble declaration list> ; <preamble declaration>

<preamble declaration> ::= <requirement declaration>
|<type declaration>|<variable declaration>
|<constant declaration>

<subsystem description list> ::= <subsystem description>|<subsystem description list> ; <subsystem description>

<specification> ::= <subsystem description list>
end

Example

- Requirement transduction sort_descend; input n, sort_array; output sort_array end;

Variable sort_array:array [1..1000] of real;
for sort_descend;
subjectto sort_array[i] >0.0 forall i = [1..n-1] ;
n:1..1000; for sort_descend;

Module sort;

fulfills sort_descend;
accesses card_reader, line_printer;
creates n, sort_array;
modifies sort_array using n, sort_array;

```
satisfies    sort_array[i] >= sort_array[i+1]
             forall i = [1..n-1]

End

End
End ;
```

Semantics

Each subsystem must have a requirement declaration that contains at least one transduction identifier and one output variable. There must also be at least one module description. The first subsystem declared (called the "main" subsystem) does not have a subsystem identifier; all others must have a unique identifier. The scope of the subsystem identifier is the entire specification.

The nonterminal symbol <specification> is the distinguished symbol of the SSL grammar.

2.1.3 Example

The example of this section was selected to demonstrate both the descriptive level of SSL and as many language elements as possible. The requirement of the problem may be stated as follows [8]:

> "A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer where it is to be processed. The words in the telegram are separated by sequences of blanks and each telegram is delimited by the word 'ZZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZZ' and 'STOP' are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word."

To complete the problem statement, several assumptions are necessary. The following alternatives were selected for the purpose of this exposition:

- The character stream from which the telegrams are constructed resides on a drum having fixed length records; the record length itself is left as an implementation option.
- The chargeable word count is the value to be printed and overlength words count as one word.
- If a physical end of file is encountered before the logical end of the data stream, an error message and the partial telegram is printed.

The software is organized into four modules as indicated by Figure 2-2. The purpose of each module is given in Table 2-7. Figure 2-3 contains the SSL description of the telegram processor. The right margin of the statement listing contains reference notes to subsections containing detailed descriptions of the language elements used.

A careful examination of Figure 2-3 will indicate an interesting application of the subsystem capability. The subroutines GET_CHAR and FILL_BUFFER occupy a separate subsystem with the sole purpose of handling file I/O. The characteristics of the device on which the telegrams are stored are encapsulated within these two modules.

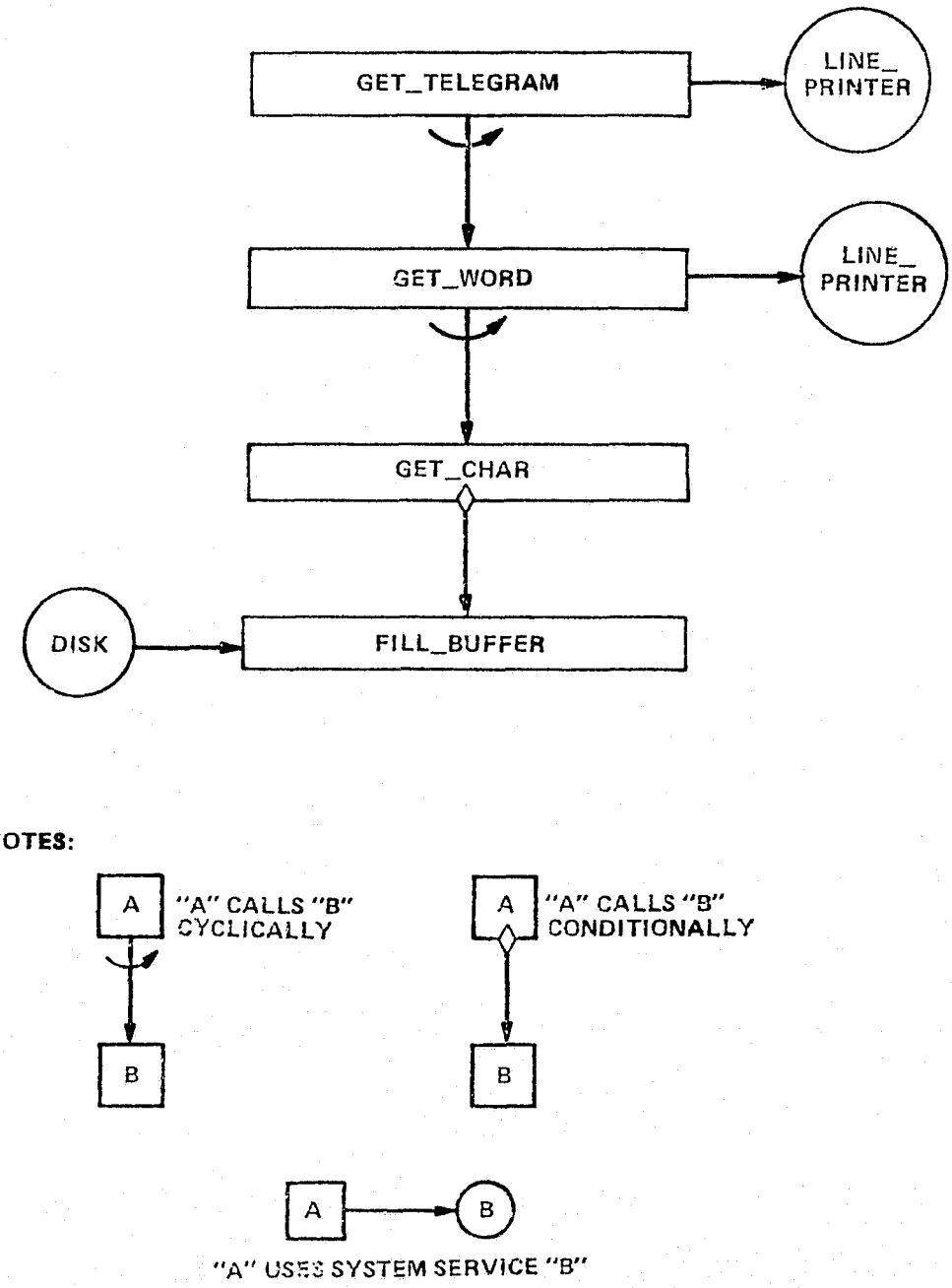


Figure 2-2 Module Structure Chart for Example

TABLE 2-7 MODULE DESCRIPTIONS FOR EXAMPLE

MODULE	PURPOSE
GET_TELEGRAM	Collects words belonging to each telegram and prints them in a neat manner along with the chargeable word count.
GET_WORD	Collects characters into words and prints error messages denoting over-length word or physical record end of file.
GET_CHAR	Returns the next character in the telegram file.
FILL_BUFFER	Enters the next physical record from the drum into the character buffer.

```
/* beginning of main subsystem preamble */                          ---- 2.1.2.3

requirement
        transductions
        collect in print;
        output
        telegram, charge_count                                      ---- 2.1.2.5
        end;

variable telegram:text;
        charge_count:integer;
            for print;
            subjectto charge_count ≥0                               ---- 2.1.2.6
        word_count:integer;
            for print;
            subjectto word_count ≥ charge_count;                    ---- 2.1.2.6
        word:array [1..12] of char;                                 ---- 2.1.2.6.2.1
            for print;
        eof_flag:boolean;                                           ---- 2.1.2.6.1.1
            for print

end; /* end of main subsystem preamble */

/* main routine  to collect words and */
/* print telegram with chargeable word count*/

module get_telegram:
        fulfills print;
        creates telegram, charge_count using word;
        creates word_count;
        modifies word_count;
        uses eof_flag;
        accesses line_printer;
        executes cyclically get_word;
        satisfies
            eof_flag or word_count = 0
        end;                                                        ---- 2.1.2.10

/* subroutine to collect characters into */
/* words */

module get_word:                                                    ---- 2.1.2.10
        fulfills collect;
        executes cyclically i_o.get_char(a_char:char;eof_flag);      ---- 2.1.2.10.1
        creates word, eof_flag;
        accesses line_printer /*prints error messages */           ---- 2.1.2.10.4
        end

end; /* end of main subsystem */
```

Figure 2-3 SSL Description for Example

```
/* beginning of i_o subsystem preamble */

subsystem i_o;                                                    2.1.2.11

requirement
        input character_file;                                     2.1.2.5.1
        transductions                                             2.1.2.5.2
        read in separate;
        output a_char, eof_flag                                   2.1.2.5.1
        end;

/* parameterize record length */
constant record_length = integer;                                 2.1.2.7

type character_record = array [1..record_length] of char;         2.1.2.6

varinble character_file:sequence of character_record;             2.1.2.6.2.5
            for read;
        buffer:character_record;
            for separate;
        a_char:char ;
            for separate;
        char_index:1..record-length;                              2.1.2.6.1.3
            for separate;
        eof_flag:boolean;
            for separate                                          2.1.2.6

end; /* end of subsystem preamble */
```

```
/* subroutine to fetch next */
/* character from file */

entry get_char (a_char; eof_flag) ;                               2.1.2.10.1
        fulfills separate;                                        2.1.2.10.3
        executes conditionally fill_buffer;                       2.1.2.10.7
        modifies char_index;                                      2.1.2.10.6
        creates a_char using buffer [char_index] , eof_flag;      2.1.2.10.2
        creates character_file, char_index;                       2.1.2.8.2
        satisfies  eof_flag implies a_char = buffer [char_index]
        end;

/* subroutine to fetch next physical */
/* record from character file */

module fill_buffer;
        fulfills read;                                            2.1.2.10.3
        assumes char_index = record_length;                       2.1.2.10.2
        accesses disk;
        creates buffer, eof_flag using character_file@ ;          2.1.2.8.2
        satisfies
            eof_flag implies buffer = character_file@             2.1.2.9.2
        end

end /* end of subsystem */
end; /* end of specification */
```

Figure 2-3 SSL Description for Example (continued)

2.2 SEMANTIC DESCRIPTION OF SSL

Our semantic description of SSL is in terms of sets and set functions grouped into n-tuples. The initial construct is the Vertex Correlation Tuple which forms the basis for definition of a single subsystem of the software structure. This tuple consists of a set of nodes, some of which have module names attached. Virtual nodes do not represent program modules, but evocations of entry nodes of separate subsystems. Modules are represented by concrete nodes. Special types of concrete nodes, called entry nodes, are a distinctive characteristic of subsystems other than the main subsystem and have special properties. First, they must have attached module names, so they can be referenced by other subsystems. Second, (as we shall see in the Requirements Graph) their predecessor may only be the root node of the subsystem which is not permitted an attached module name.

Vertex Correlation Tuple

$$\chi = (N, C, V, G, Z, M, mod)$$

where

(1) N = a finite set of labeled nodes

(2) C = a subset of N called concrete nodes

(3) V = a subset of N called virtual nodes; $V \wedge C = \Phi$

(4) G = a subset of C called entry nodes

(5) Z = a distinguished node called the root

(6) M = a finite set of module names

(7) $mod : N \rightarrow M$: a module name mapping function

(a) If $n \varepsilon C$, then cardinality (mod (n)) = 0 or 1.

(b) If $n \varepsilon G$, then mod (n) $\neq \Phi$; entry nodes are required to have an attached module name.

(c) If $n \varepsilon V$, then mod (n) $\neq \Phi$; virtual nodes are required to have an attached module name.

(d) If $G \neq \Phi$, then mod (Z) = Φ; the root nodes of subsystems other than the main subsystem are not permitted to have attached module names.

The module mapping function (mod) implies:

(1) Not all concrete nodes have an associated module name.

(2) Module names associated with virtual nodes are not necessarily unique; each evocation of an entry node is represented by a distinct virtual node.

The nodes in χ will be collected into a weakly-connected digraph based on a mapping of nodes to subsets of requirements. (For reasons of clarity and stand-alone interpretation, we are using "requirements" in the sense we used "transductions" in the previous part.) To accomplish this, the requirements are partially ordered by implication. If R is a set of requirements, r_1, r_2 εR, we write $r_1 \leq r_2$ (read "r_1 is implied by r_2"). For example, consider a line editor. The requirements are:

edit, search_file, delete_line, add_line .

The implications between these requirements are as follows:

(1) delete_line $\leq$ edit

(2) add_line $\leq$ edit

(3) search_file $\leq$ edit

(4) Search_file $\leq$ delete_line

(5) search_file $\leq$ add_line

(6) delete_line $\not\leq$ add_line, add_line $\not\leq$ delete_line: i.e., {delete_line, add_line} is unordered.

With nodes mapped onto subsets of a set of partially ordered requirements, it is possible to define the predecessor relationship between nodes by a function. The function (called simply the predecessor function) requires that $r_2 \leq r_1$ in order for n_1 to be a predecessor of n_2 where r_1 is a certain requirement attribute of n_1 depending on r_2.

The Requirements Graph

$\Pi = (\chi, R, B, req, pr)$

where

(1) χ = A Vertex Correlation Tuple

(2) R = a finite set of labeled requirements, partially ordered by implication

(3) B = a base set of R; if $r_1 \varepsilon B$, then $\nexists\, r_2$, $r_2 \varepsilon R$, $r_1 \neq r_2$ such that $r_1 < r_2$

(4) $req: N \rightarrow 2^R$; = the requirements mapping set function; a subset of R is associated with each node

(a) $req(Z) = B$

(b) If $n \varepsilon N$ then $req(n) \neq \Phi$

(5) $pr: N \rightarrow 2^N$ = the predecessor set function

(a) $pr(Z) = \Phi$; if $n \neq Z$ then $pr(n) \neq \Phi$

(b) If $n_1 \varepsilon pr(n_2)$ then we define (n_1, n_2) to be adjacent in a digraph sense

(c) If $r_2 \varepsilon req(n_2)$, $n_1 \varepsilon pr(n_2)$ then there exists $r_1 \varepsilon req(n_1)$ such that $r_2 \leq r_1$

(d) If $n_1, n_2 \varepsilon pr(n_3)$, $r_1 \varepsilon req(n_1)$, $r_2 \varepsilon req(n_2)$ then $r_1 \not\leq r_2$, $r_2 \not\leq r_1$, i.e., the requirement attributes if taken one each from the predecessors of a node are unordered

(e) If $n_1 \varepsilon pr(n_2)$ then $n_1 \varepsilon C$; i.e., virtual nodes may not have successors

(f) If $n_3 \varepsilon V$, $n_1 \varepsilon pr(n_3)$, $n_2 \varepsilon pr(n_3)$ then $n_1 = n_2 \neq \Phi$; i.e., virtual nodes have exactly one predecessor

(g) If $n_1 \varepsilon pr(n_2)$, $n_2 \varepsilon V$ then $req(n_1) = req(n_2)$; i.e.,

virtual nodes have the same requirement attributes as their predecessor

(h) If $n \varepsilon G$ then $pr(n) = \{Z\}$

In addition to modules at graphical nodes, the other resources available to a software system are environment objects and data objects. Data objects are created at one and only one point within the system for use elsewhere; environment objects are not "created" but may be used or "accessed" at various points within the structure. A further distinction is that data objects, but not environment objects, have requirement attributes. The requirement attributes of data objects are not necessarily the same as those of the creating node. These attributes delimit where within a structure a data object may be used.

Object Distribution Graph

$$\Lambda = (\pi, Q, E, cr, drq, acc)$$

where

(1) π = A Requirements Graph

(2) Q = DUL a finite set of labeled data objects; $D \cap L = \phi$; if $d \varepsilon D$ then d is called a global data object; if $d \varepsilon L$ then d is called a local data object

(3) E = a finite set of labeled environment objects

(4) $cr: C \to 2^D$ = object creation set function

(a) If $d \varepsilon D$, $d \varepsilon cr(n_1)$, $d \varepsilon cr(n_2)$ then $n_1 = n_2$; i.e., an object is created at one and only one node

(b) If $d \varepsilon D$ then $d \varepsilon cr(n)$ for some n: i.e., all data objects are created within the system

(5) $drq: D \to 2^R$ = data requirements set function

(6) $acc: C \to 2^E$ = environment access set function; $e \varepsilon acc(n)$ if $e \varepsilon E$ and e is accessed at concrete node n

Thus far, the constituents of a software structure that have been developed are:

(1) Nodes with attached modules
(2) Node interconnections based on requirements ordering
(3) Data objects created and environment objects accessed.

In order to complete the interconnection graph, it is necessary to depict the utilization of data objects. A global data object is assigned requirement attributes via the drq function and these attributes are compared to those of potential user nodes. If n is a concrete node then d, a data object, is available for use only if for each $r_1 \varepsilon req(n)$ there exists $r_2 \varepsilon drq(d)$ such that $r_1 \leq r_2$ or d is a member of the release set of n or an immediate successor of n. In effect, data object requirement attributes place an upper bound on the generality of modules permitted use as well as limit the object's scope to specific segments of the graph.

Subsystem Structure Graph

$$\Sigma = (\Delta, rel, av, use)$$

where

(1) Δ = an Object Distribution Graph
(2) $rel:(N,N) \rightarrow 2^Q$ = release set function;
if $rel(n_1,n_2) \neq \Phi$ then $n_1 \varepsilon pr(n_2)$; i.e., n_1 and n_2 must be adjacent in a digraph sense if n_1 may release objects to n_2
(3) $av:N \rightarrow 2^Q$ = availability set function:
(a) If $d \varepsilon D$, $n \varepsilon C$, $d \varepsilon av(n)$, $r_1 \varepsilon req(n)$ then $r_1 \leq r_2$ for some $r_2 \varepsilon drq(d)$; i.e., each requirement attribute of a concrete node must be subsumed by at least one of the requirment attributes of the global data object.

(b) If $d \varepsilon L$, $n_1 \varepsilon N$, $d \varepsilon av(n_1)$ then either $d \varepsilon rel(n_1,n_2)$ or $d \varepsilon rel(n_2,n_1)$; i.e., local data objects may be shared only by pairs of nodes that are adjacent in a digraph sense.

(c) If $d \varepsilon av(n)$, $n \varepsilon V$ then $d \varepsilon rel(n_1,n)$ where $pr(n) = \{n_1\}$; i.e., a data object available at a virtual node must have been released by its parent.

(d) If $d \varepsilon rel(c,v)$ then $d \varepsilon av(c)$ where $c \varepsilon C$, $d \varepsilon Q$, i.e., concrete nodes may release to virtual nodes only those data objects which are elements of the availability set of the concrete node.

(4) $use: N \rightarrow 2^D$ = usage set function; $d \varepsilon use(n)$ if $d \varepsilon av(n)$ and d "is used" at n as determined by the designer.

The definition of the System Structure Graph is the final step in describing the semantics of SSL. The ultimate goal is an automated version of SSL which will:

- Permit partial system definition with consistency checking at the appropriate level of detail, and
- Participate in the design by supplying derived information and indicating the next logical steps in an incremental development.

Accomplishing this will depend on discerning higher order relationships among the functions describing SSL. This involves identifying theorems and algorithms that derive the set defined by one function through the sets defined by other functions. A subsequent section represents our efforts in this direction.

2.3 OVERVIEW OF THE SSL TRANSLATOR

In this section we give the formal software requirement for the translator, the functional design, and notes on detailed design. Significantly more detailed information exists on the functional design, including a description in the SSL language, than that presented here. The format and notation used for the functional design description in this report is more conventional.

The notes on detailed design concentrate on the more critical semantic and structure analysis phase. Included are:

- A method for determing the legitimacy of data/module interconnections
- A method for ordering modules in a manner that facilitates module interconnection and recursive analysis
- A method for constructing a matrix over which module interconnection and recursive analysis can be performed
- Rules for determining which modules are potentially recursive.

2.3.1 The Formal Software Requirement

In our terminology, the software requirement is a detailed declaration of the actions to be performed by a single software package. It includes what is to be done as well as initial estimates of "when" and "how well" various actions are to be performed. The constituent parts of the software requirement decomposition are:

Direction - A general statement of the boundaries of the problem

Input - Data or documents available to the software system or subsystem from external sources

Transductions - A list of processing steps to be performed, each of which translate a stimulus into a response

Output - Data or documents produced by the software for external purposes

Constraints - A list of capacities, design objectives, or resources to be observed

Preconceptions - A list of specific design alternatives to be observed

Implications - A binary relation existing between certain pairs of transductions, indicating which are substeps of others.

Of the seven divisions, the constraints and preconceptions are optional. If all transductions are independent, the implications may also be omitted.

The level of detail required for the SRD is generally greater than could be expected of a casual user of data processing services. For this reason, it is expected that the SRD will be completed by an experienced computer specialist after reviewing the user requirments. Completing the SRD is the beginning step of functional design.

Figure 2-4 depicts the SRD for the SSL translator. There are three major segments within the transductions: program analysis, structure analysis, and report summarization. The implication relations between tranductions is represented in Figure 2-4 by indentation within the transduction division rather than within a separate implications division.

The only system (or requirement) level input is the SSL source language. System output consists of eight items. In addition to the source listing, syntax and semantic errors, these consist of:

Module concordance - An alphabetical list of all modules; for each module the following information is given: 1) modules it calls, 2) modules which call it, 3) variables referenced, 4) environmental objects referenced, and 5) requirement attributes

Variable concordance - An alphabetical list of all variables; for each variable the following information is given: 1) requirment attributes and 2) modules which reference it

Requirement concordance - An alphabetical list of all requirment transductions and constraints; for each, the following information is given: 1) modules to which attached as an attribute, 2) variables to which attached as an attribute, 3) (for transductions) the transductions immediately above and below it in a partially ordered sense

Summary - A summarization of the number of modules, variables, errors, etc. by subsystem

Index - A cross reference guide to facilitate access to parts of the SSL generated report

DIRECTION

Implement an automated translator for Software Specification Language (SSL).

INPUT

SOURCE: A set of logically connected SSL statements.

TRANSDUCTIONS

TA_PROG_ANALYSIS: Analyze the SSL program
TB_SYNTAX: Analyze the program syntax
TC_TOKEN: Reduce the next lexical token
TD_CARD: Read and print the next card image
TC_SYN_ERR: Perform syntax error recovery procedures
TC_TABLES: Construct dynamic tables
TD_REQ_ST: Process requirement statement
TD_CONS_ST: Process constant statement
TD_FUL_ST: Process fulfills statement
TD_TYPE_ST: Process type statement
TE_RECORD: Process record and digital forms

Figure 2-4. SRD for SSL Translator

```
                TF_ARRAYS:  Process array forms
                TF_SUBRANGE:  Process subrange forms
                TF_SCALARS:  Process scalar forms
                TF_BASIC:  Process basis type forms
                TF_POINTER:  Process pointer forms
           TE_FILE:  Process file and sequence forms
                TF_ARRAYS, TF_SUBRANGE, TF_SCALARS, TF_BASIC, TF_POINTER
      TD_VAR_ST:  Process variable statement
           TE_FOR:  Process for clause
           TE_POLISH:  Analyze a polish string
           TE_RECORD, TE_FILE
      TD_MOD_ENT_ST:  Process module and entry statements
                TF_BASIC
           TE_RELEASE
      TD_SUBSYS_ST:  Process subsystem statement
      TD_EXEC_ST:  Process executes statement
           TE_RELEASE:  Process release form
                TF_BASIC
      TD_USES_ST:  Process uses statement
           TE_DATA_LIST:  Add items to data list
      TD_MOD_CR_ST:  Process modifies or creates statement
           TE_DATA_LIST
      TD_XMIT_RX_ST:  Process transmit or receives statement
           TE_DATA_LIST
      TD_ASUM_SAT_ST:  Process assumes and satisfies statements
           TE_POLISH
```

Figure 2-4. SRD for SSL Translator (continued)

```
TA-STRUC_ANALYSIS:   Analyze the software decomposition structure
     TB_ELEMENTS:  Ascertain all elements are consistent within themselves
          TC_SUB_DEF:  Evaluate all subsystem definitions
          TC_MOD_DEF:  Evaluate all module definitions
          TC_VAR_DEF:  Evaluate all variable definitions
          TC_TYPE_DEF:  Evaluate all type definitions
          TC_REQ_DEF:  Evaluate all requirement definitions
     TB_SETS:  Construct and evaluate all interelement relationships
          TC_REQ_VS_REQ:  Ascertain requirement definitions are consistent
               TD_REQ_CON:  Construct requirement concordance lists
                    TE_MDR_MATRIX:  Construct a module/data/requirement matrix
                    TE_NEXT_REQ:  Find next alphabetically ordered requirement
                    TE_MOD_ATT:  Construct list of modules to which requirement
                                 attached
                    TE_SUP_ATT:  Construct list of superstep transductions
          TC_MOD_VS_REQ:  Ascertain module hierachy consistent with requirements
               TD_MOD_VECT:  Construct a module requirement vector
               TD_CLOS_VECT:  Perform closure over a requirement vector
          TC_MOD_VS_DAT:  Ascertain modules use data consistent with requirements
               TD_AD_VECT:  Construct a variable availability vector
               TD_MOD_VECT, TD_CLOS_VECT
               TD_DAT_CON:  Construct data concordance list
                    TE_DAT_MATRIX:  Construct data concordance list
                    TE_NEXT_DAT:  Find next alphabetically ordered variable
                    TE_REFER:  Construct list of modules reverencing
                               data object
```

Figure 2-4. SRD for SSL Translator (Continued)

TC_MOD_VS_REL: Ascertain module release sets are consistent

TC_MOD_VI_MOD: Ascertain module calling hierarchy consistent

TD_ORDER: Order modules by forward paths

TD_MATRIX: Construct D^+ Matrix

TD_RECURSIVE: Perform recursive analysis

TE_POT_RECUR: Mark recursive modules

TE_LAT_HEAD: Mark latch and head modules

TD_INV_HIER: Construct inverse hierarchial list

TE_INV_MOD: Find next inverse call module

TC_REPORT_GEN: Generate report

TD_REQ_REPORT: Print requirement concordance

TD_DAT_REPORT: Print variable concordance

TD_MOD_REPORT: Print module concordance

TA_REPORT: Summaries and index report

TB_SUMMARY: Print summary of software report

TB_TOC: Print table of contents for report

TC_MOD_LIST: Print modules in alphabetical order

TD_NAME_PAGE: Print a single name and page number

TC_DAT_LIST: Print variables in alphabetical order

TD_NAME_PAGE

TC_REQ_LIST: Print requirements in alphabetical order

TD_NAME_PAGE

OUTPUT

SOURCE_LISTING: A line printer listing of the original SSL source

SYN_ERRORS: SSL syntax error diagnostics interleaved with the source listing

Figure 2-4. SRD for SSL Translator (continued)

SEM_ERRORS: A printed summary of all semantic incongruities

HIER_LIST: A list (alphabetical) of all modules which includes modules referenced, inverse hierarchy, data referenced, environment objects referenced, and requirement attributes.

DATA_LIST: A list (alphabetical) of all variables accompanied by the names of modules which use them and requirement attributes assigned.

REQ_LIST: A list (alphabetical) of all requirements cross referenced with modules, variables, and other requirements

SW_SUMMARY: A summarization of the software which includes counts for modules, variables, errors, etc.

INDEX_LIST: A cross reference guide for facilitating access to parts of the SSL generated report

IMPLICATIONS

(Implications are represented by indentation within the transductions substation.)

CONSTRAINTS

LANGUAGE: ANSI FORTRAN IV will be used to implement the translator

HOST_MACHINE: The translator will be written in a manner amenable to transportability; the host machines shall at least include the IBM S360/65 and Univac 1108.

Figure 2-4. SRD for SSL Translator (continued)

2.3.2 Functional Design Overview

The module decomposition follows closely the requirement decomposition of Figure 2-4. There are three phases. The first phase analyzes the SSL source input and constructs a hierarchical file containing all object attributes and interrelationships. The second phase analyzes the information within the hierarchical file for semantic errors, then generates a report on the software organization. The third phase prints a summary and generates an index to the report.

Figure 2-5 depicts a high level view of the software organization. The three phases depicted are executed once consecutively to produce the report from the SSL source program. The block labeled "SSL Translator Main Program" is the control program. The actions performed by each of the phases is discussed in the paragraphs below.

The first phase (controlled by the block labeled "Syntax Analyzer" in Figure 2-5) is the source program analysis phase. Its function is to read the source program and construct the file used in subsequent phases. The purposes of the principal blocks are as follows:

- Syntax Analyzer - The subroutines of this block parse input source statements, emit syntax diagnostics and pass the parse trees (polish notation) to the semantic analyzer
- Lexical Analyzer - The subroutines of this block read and print the source statements; from the source statements, the tokens (numbers, reserved words, delimeters, and identifiers) are collected and returned to the syntax analyzer
- Semantic Analyzer - The subroutines of this section collect the parse trees and, when one is sufficiently complete, passes it to lower level

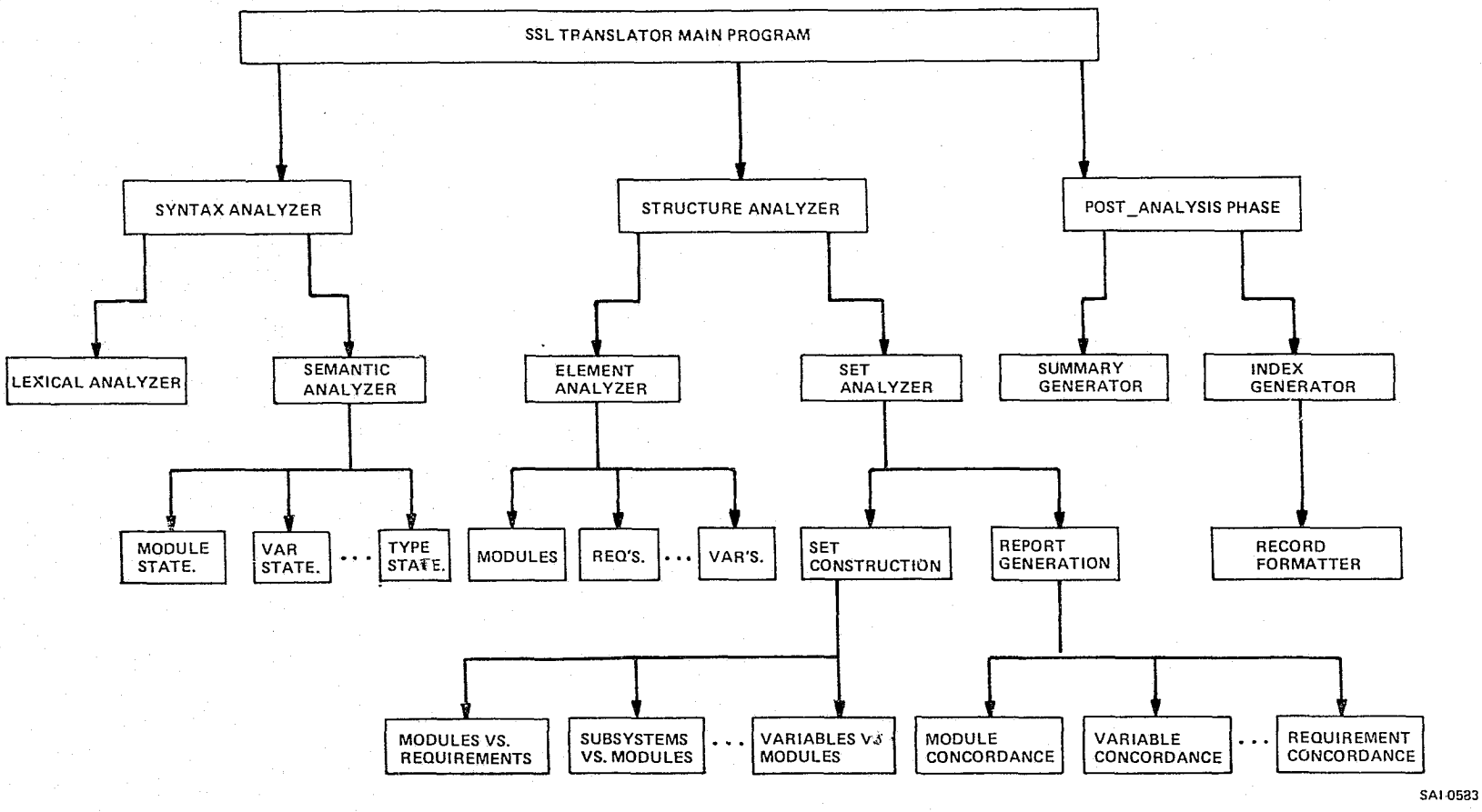


Figure 2-5. Block Diagram for SSL Translator

subroutines; the lower level subroutines called by the semantic analyzer are differentiated by statement type and each constructs a specific part of the hierarchical file.

The second phase (controlled by the block labeled "Structure Analyzer" in Figure 2-5) is where the software interconnections are examined for consistency. Each element (subsystem, module, variable, etc.) is examined first for internal or self-consistency. Self-consistency includes each element being defined, referenced, and having all attributes assigned. After element analysis, set analysis takes place. Set analysis involves testing the consistency of all interelement references. This task is performed in two parts. The first part constructs the interelement relationships in the form of a set of boolean matrices. Semantic error analysis is carried out from the matrices and their representation is converted to a list structure. This list structure is used in the second part of set analysis to generate the software structure report.

The role of the second phase can be further clarified by examining the function of each of the blocks in Figure 2-5.

- Structure Analyzer - Control routine for phase 2
- Element Analyzer - On a subsystem by subsystem basis, the subroutines of this block examine the various elements that comprize the subsystem for intraelement consistency; the lower level subroutines that the element analyzer calls are differentiated on the basis of element type.
- Set Analyzer - Control routine for set analysis.

- Set Construction - The subroutines of this block construct a data base containing interelement dependencies, and alalyze the dependences for semantic errors; the lower level subroutines called by set construction are differentiated on the basis of element type pairs (modules vs. requirements, subsystems vs. modules, etc.).

- Report Generation - The subroutines of this block generate the software structure report; the lower level subroutines called by report generation are differentiated on the basis of the various sections of the report (modules, variables, etc.).

The third phase (controlled by the block labeled "Post-Analysis Phase" in Figure 2-5) summarizes and prints an index for the report previously generated. The major blocks of this phase may be summarized as follows:

- Post-Analysis Phase - Control routine for the third phase.

- Summary Generator - Prints a summary of the software structure such as the number of modules, number of variables, and number of errors per subsystem.

- Index Generator - The subroutines of this block generate a index list for each module, variable, etc., including which page of the source listing and concordance listings it occurred.

- Record Formatter - Prints a single line of the index which includes a name with page numbers.

2.3.3 Detailed Design Notes

Phase 1 of the translator is a standard parser combined with data structure synthesis routines. Phase 3 is simply an output editor. The crucial subset is phase two in which the

interelement relations are analyzed semantically. The purpose of this section is to expound both algorithmically and theoretically on some of these relationships.

2.3.3.1 Assessing Data Availability

Recall that requirement attributes are attached to both modules and data objects as a means of providing requirements traceability. A secondary effect of requirement attributes is that they limit the availability of data objects; i.e., a data object may not be used at a concrete node unless all the requirements attributes of its module are equal to or implied by requirement attributes of the data object. Therefore, one might expect a close relationship between the requirement attribute functions (req, drq) and the availability function (av). This relationship is expounded below.

Let $R = \{r_1, r_2, \ldots, r_k\}$ be a set of requirements. Let $P(d) = (p_1, p_2, \ldots, p_k)$ be a data object requirement vector where:

$$p_j = \begin{cases} 1 & \text{if } r_j \in drq(d) \text{ or } r_j < r_i \text{ where } r_i \in drq(d) \\ 0 & \text{otherwise.} \end{cases}$$

Let $Q(n) = (q_1, q_2, \ldots, q_k)$ be a node requirement vector where:

$$q_j = \begin{cases} 1 & \text{if } r_j \in req(n),\ n \in C \\ 0 & \text{otherwise.} \end{cases}$$

Theorem

For any $n \in C$, $d \in av(n)$ if and only if $\sum_{i=1}^{k} q_i = P(d) \cdot Q(n)$.

Proof (necessary) assume $n \in C$ and $d \in av(n)$. Then, by definition of the av function, for any $r_1 \in req(n)$ there exists an r_2 in drq(d) such that $r_2 \geq r_1$. So $q_j = 1 \Rightarrow r_j \in req(n) \Rightarrow r_j \leq r_k$ for some $r_k \in drq(d) \Rightarrow p_j = 1$; $\therefore$

$$\sum_{i=1}^{k} q_i = P(d) \cdot Q(n).$$

The sufficiency part of the proof is carried out similarly.

END OF PROOF

Note that the theorem applies only to concrete nodes since virtual nodes, unlike concrete nodes, depend additionally on the release function (rel).

2.3.3.2 Assessing Consistency of Data Usage

Data object usage at a module is dependent upon its availability at that module. However, the two sets are derived from different perspectives and require cross checking. Anomalies should prompt the designer to re-think the requirement attributes assigned objects, a healthy exercist.

Let's begin by recalling that a data object, d, is not eligible to be used at a module, n, unless $d \varepsilon av(n)$. Let $D = \{d_1, d_2, \ldots, d_m\}$ be the set of data objects within the system. For some node n, let $U(n) = [u_1, u_2, \ldots, u_m]$, the usage vector, be a vector where

$$u_i = \begin{cases} 1 & \text{if } d_i \varepsilon use(n),\ n \varepsilon C \text{ where } C \text{ is concrete node} \\ & ;\ i = 1, 2, \ldots, m \\ 0 & \text{otherwise.} \end{cases}$$

Let $W(n) = [w_1, w_2, \ldots, w_m]$, the candidate vector, be a vector where

$$w_i = \begin{cases} 1 & \text{if } d_i \varepsilon av(n),\ n \varepsilon C;\ i = 1, 2, \ldots, m \\ 0 & \text{otherwise.} \end{cases}$$

Given U(n) and W(n) the usage set assigned to node n is legitimate only if

$$u_i = u_i \cdot w_i ; \quad i = 1, 2, \ldots, m$$

Furthermore, if the set is illegal, the culpable object is identified by the element of U(n) for which the above test fails.

2.3.3.3 Ordering Modules for Analysis

The predecessor relation defined in SSL semantics only partially orders the nodes (modules). There exists more than one total order that adequately reflects the partially ordered properties of any nontrivial module set. It is necessary to select a (total) ordering algorithm in order to perform analysis in a deterministc manner.

The algorithm preserves the natural partial order of the modules. Definition of the following terms are necessary:

e	A unique module called the root or entry module (of a subsystem)
ord(m)	The order number of module m; initially zero for all modules
pr(m)	The predecessor function for module m as defined in the SSL semantics; $pr^{-1}(m)$ is the successor function
n	The set of modules
#S	The cardinality of the set S

The algorithm is as follows:

(1) Let S_1 = e ; set ord (e) = 1

(2) Let p = 1, k = 1

(3) If $m_p \in S_k$ and there exists $m_q \in N \cap \{pr^{-1}(m_p) - S_k\}$, then

  (a) For each m_r such that $p < ord(m_r) \leq \#S_k$, increase ord (m_r) by 1

  (b) Define ord (m_q) = p + 1, $S_{k+1} = S_k \cup \{m_q\}$

  (c) Increase p and k each by 1

  (d) Return to step (3).

(4) If p >1, decrease p by 1 and return to step (3); otherwise stop.

This algorithm assigns an order number to each module. Furthermore, the order numbers increase monotonically along forward (non-recursive) calling paths. Figure 2-6 illustrates the order number assignments for an arbitrary block diagram. Note that more than one order assignment combination fulfills the criterion of increasing order numbers along forward paths.

2.3.3.4 Construction and Closure of Dependency Matrices

A dependency matrix (or adjacency matrix) is an n x n boolean matrix where there are n modules. Rows and columns must be ordered equivalently to the order numbers acquired from the algorithm given above. The elements of the dependency matrix, D, are as follows:

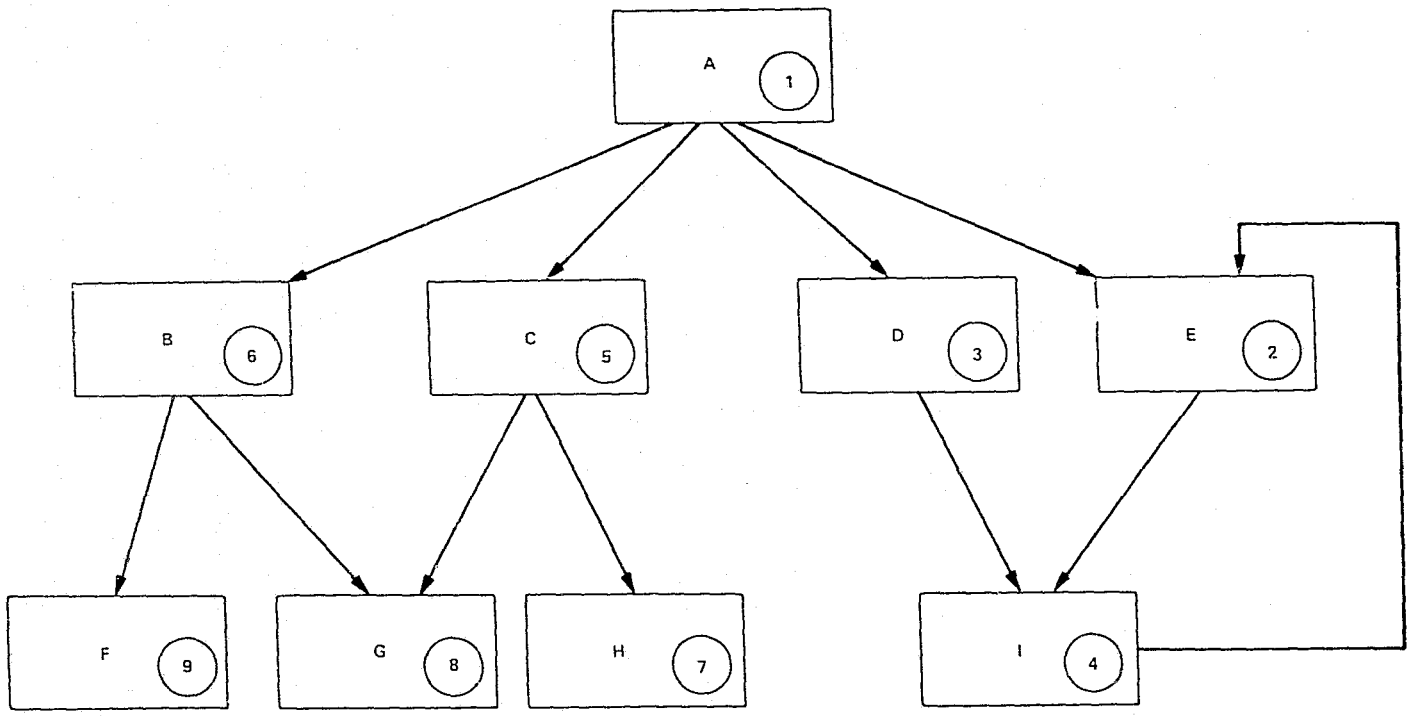
$$d_{ij} \begin{cases} = \text{ true if module of order i references module of order j} \\ = \text{ false otherwise} \end{cases}$$

Once constructed, the rows of D yield "called" lists and the columns "called by" lists.

Closure of D - In the closure of the matrix D (denoted D^+) an element d_{ij} will be true if there exists a sequence in D, d_{ip}, d_{pq} ,d_{qr}, . . .,d_{sj}, all of which are true.

One way of deriving D^+ is by raising D to the n^{th} power. The algorithm given here is much more efficient.

```
D: array [1 . . n, 1 . . n] of boolean;
i, j, k: integer;
for j = 1 to n do
    begin
    for i = 1 to n do
      if D [i, j] and i ≠ j then
          for k = 1 to n do
              D [i, k] = D [i, k] or D [j, k]
    end
end;
```



SA-0072

Figure 2-6. A Module Ordering Example

2.3.3.5 Recursive Analysis Using Dependency Matrices

It has already been noted that "called" and "called by" lists are explicitly represented in D. What remains is the derivation of the recursive information required for the module concordance of the previous section. Specifically, the determination of head modules, latch modules, and potentially recursive modules. (A latch module, in the context of recursive analysis, is one that makes a recursive call. A head module is one to which a recursive call is made).

Theorem

If d_{ij} = true and $j \leq i$ then
i is the order no. of a latch module and j is the order no. of a head module of a recursive subsystem.

Proof

Assume ord(m_1) = i and ord(m_2) = j. Since d_{ij} = true, m_1 calls m_2. If there exists a forward calling path from m_2 to m_1 then the call of m_2 by m_1 is clearly recursive with m_1 being the latch and m_2 being the head. So, assume there is no forward path from m_2 to m_1. If there does not exist a forward path from m_2 to m_1 then the path (m_1, m_2) is a forward reference. This implies ord(m_1) <j, contradicting the original assumption.

End of Proof

Figure 2-7 is the D matrix for the dependency chart of Figure 2-6. It illustrates that 4 (Module I) is a latch module and 2(Module E) is a head module.

Theorem

If d_{ii}^{+} = true and i = ord(m) then m is potentially recursive (i.e., there exists a path from m back to itself).

Proof

By definition, d_{ii}^{+} is true if there exists a sequence, d_{ir}, d_{rs}, ..., d_{ti}, all of which are true. This implies the existence of a reference path, m, m_1, m_2, ..., m, thus m is potentially recursive.

End of Proof

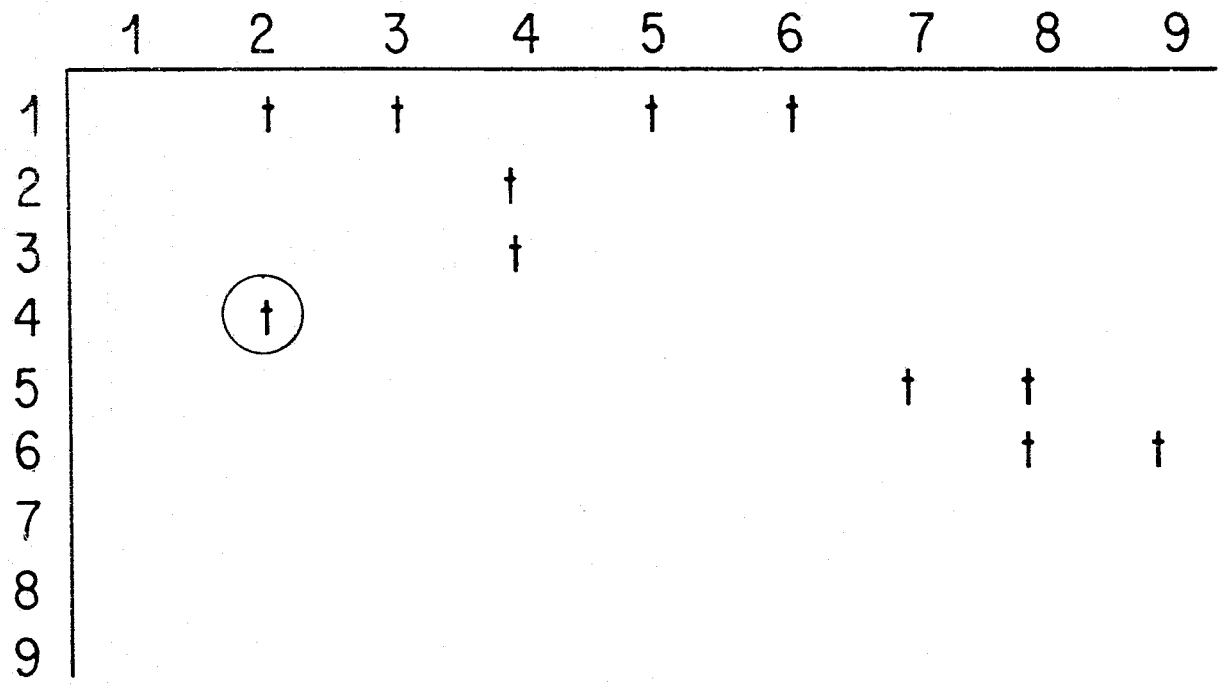


Figure 2-7. An Example D Matrix

Figure 2-8 is the D^+ matrix for the dependency chart of Figure 2-6. It illustrates that 2 and 4 (Modules E and I) are potentially recursive.

Note that the existence of a recursive path does not necessarily prove that modules on the path are recursive. During execution, the recursive path may never be traversed. Note also that a recursive call made unconditionally by a latch module is a potential infinite loop.

The techniques above were discussed in the context of the module concordance. A subset of the same methods would apply also to the data concordance and to the analysis of the requirement transductions.

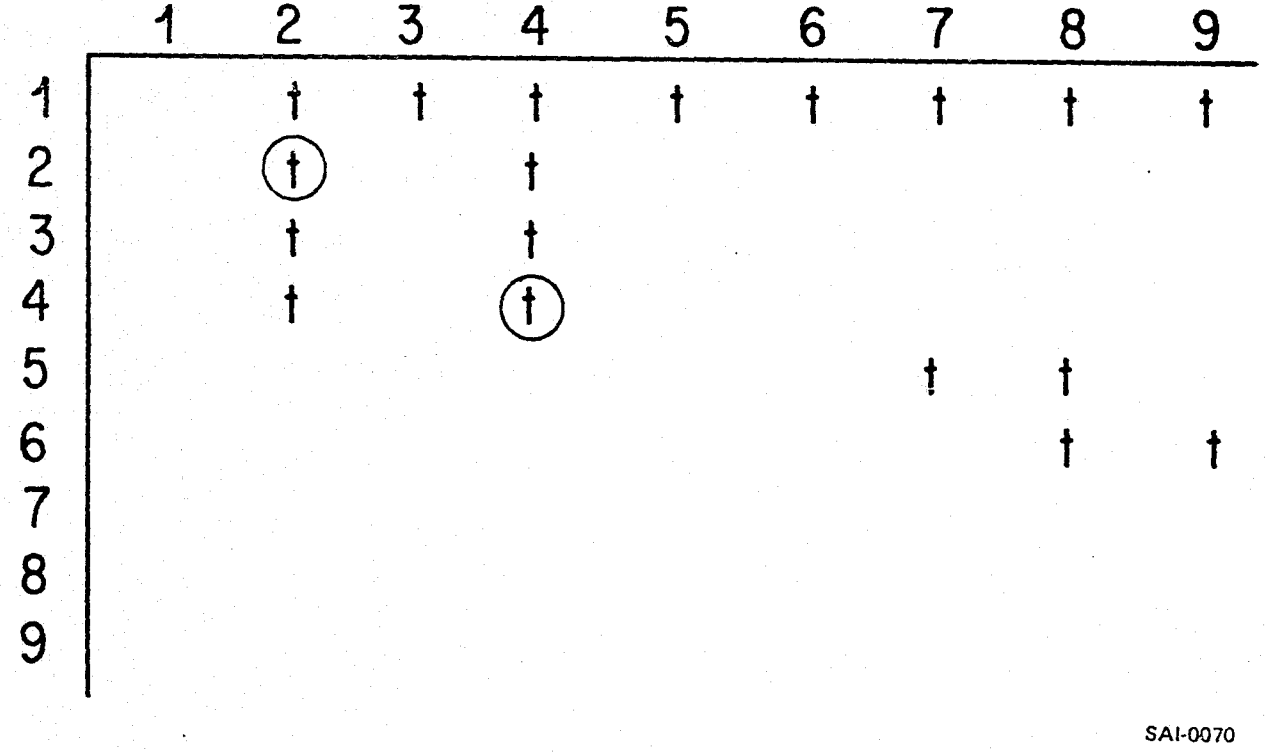


Figure 2-8. An Example D^+ Matrix

3. DATA BASE VERIFIER SUBSYSTEM DESIGN

As a result of the study and analysis conducted under SOW task Phase A, item 3, we performed a high level design (i.e., software development through the requirements and functional specification stages) of a data base verifier subsystem (DBVS). The functions of this data base verifier subsystem are analysis of the Data Manipulation Language (DML) commands within a FORTRAN source deck, collection of pertinent descriptions of the stored data base as viewed by the program(s), and printing of the subschema information in a user oriented format. The accomplishment of these functions was the goal of each step in the DBVS software design.

At the requirements stage of the development of the DBVS, we produced a Subsystem Software Requirements Document (SSRD). This document was written in accordance with the requirements methodology that we recommended as a result of analysis performed during this contract period (cf. Part I, subsection 2.1 of this report). Subsection 3.3 contains the SSRD which formed the basis of the functional design of the DBVS. For the specification stage, we used the Software Specification Language (SSL) that was designed under this contract and which is explained in Part I, subsection 2.2 and Part II, section 2 of this final report and in the special report, "SSL-A Software Specification Language."

A general description of the two main phases of the data base verifier, DML Statement Processing and Subschema Information Processing, are presented in subsections 3.1 and 3.2. These subsections are outlined in Table 3-1.

TABLE 3-1. DBVS Description Outline

1. Data Base Verifier Subsystem
   - 1.1 DML Command Processing
     - 1.1.1 DML Statement Recognition
     - 1.1.2 DML Statement Parsing
     - 1.1.3 DML Statement Components Storage
   - 1.2 Subschema Information Processing
     - 1.2.1 Subschema, realm, set, record, privacy, or error information retrieval
     - 1.2.2 Subschema, realm, set, record, privacy, or error information tabulation
     - 1.2.3 Summary Reporting
       - 1.2.3.1 Subschema, realm, set, record, privacy, or error information printing

3.1 DML STATEMENT PROCESSING

As each source statement is read, the first label (after the statement number field) of a non-comment statement is isolated by the DBVS using the information in Table 3-2. If this label is succeeded by a left parenthesis, the DBVS compares this label with the keywords contained in the FORTRAN DML Command Table (cf. Table 3-2). When a match is found, the label is entered into the keyword sequence, and parsing of the statement continues. (During parsing of the DML statements, the identified components of each clause or statement, i.e., the keywords, identifiers, or list items, are placed in three sequences. The sequence types were chosen because this type may vary dynamically in length in response to the variance of the number of keywords, identifiers, and list items within a DML statement).

Within each DML clause, identifiers (which occur to the right hand side of the equal sign) must be isolated and entered into the identifier sequence. These identifiers could contain CODASYL keywords as shown in Table 3-4 or one of the items (some of which contain keywords) in Table 3-5.

Some DML statements such as FETCH, MODIFY, etc. allow specification of list items which must be isolated and entered into the list item sequence. According to CODASYL, the items fall into the two categories described in Table 3-6.

The processing of DML statements including the construction of the keyword, identifier, or list item sequences continues in the manner described above until the entire statement has been parsed. Then the appropriate module is evoked according to the DML statement and its associated subschema information.

TABLE 3-2. CHARACTER TABLE

A	N	0	-
B	O	1	*
C	P	2	/
D	Q	3	(
E	R	4	)
F	S	5	,
G	T	6	.
H	U	7	$
I	V	8	'
J	W	9	:
K	X	blank	
L	Y	=	
M	Z	+	

TABLE 3-3. FORTRAN DML COMMAND TABLE

1. FETCH	9. ORDER
2. FIND	10. INVOKE
3. GET	11. READY
4. STORE	12. FINISH
5. MODIFY	13. ACCEPT
6. ERASE	14. USE
7. CONNECT	15. PRIVACY
8. DISCONNECT	16. QUIT

TABLE 3-4. ALLOWABLE KEYWORDS WITHIN DML STATEMENTS

1. SUBSCHEMA	14. DUPLICATE	27. INSERT
2. SCHEMA	15. ANY	28. REMOVE
3. ALL	16. OFFSET	29. STORE
4. SET	17. FIRST	30. MODIFY
5. REALM	18. LAST	31. FIND
6. UPDATE	19. NEXT	32. GET
7. RETRIEVAL	20. PRIOR	33. ERASE
8. EXCLUSIVE	21. CURRENT	34. FETCH
9. PROTECTED	22. OWNER	35. ORDER
10. CONCURRENT	23. RSE	36. CONNECT
11. ERROR	24. USING	37. DISCONNECT
12. RECORD	25. DISPLAY	38. OTHER
13. KEY	26. PRIVACY	39. STATUS

TABLE 3-5. IDENTIFIER SEQUENCE ELEMENTS

NOTE: These CODASYL definitions are predicated on the working document, "FORTREV," of the ANS committee for the proposed revised FORTRAN, (X3.9). This document was printed in the March 1976 issue of SIGPLAN Notices under the title, "Draft Proposed ANS FORTRAN."

- Character constant - is an apostrophe followed by a non-empty string of characters followed by an apostrophe.
- Character expression - is used to express a character string consisting of a character primary alone or concatenated with other character primaries. A character primary may be a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, character function reference, or character expression enclosed in parentheses (cf. FORTREV 75-09-26, Section 4).
- Character variable - FORTRAN variable of type character.
- Data base key - Integer variable. (FORTREV's definition of variable excludes array elements.)
- Data base name - record name, set name, realm name, or character expression. (The first 3 are names in the subschema being used.)
- Data base names - data base name.
- Error phrase - contains keyword error and either statement number or subroutine (with arguments if applicable) name.
- Retain - contains keyword RETAINING and either 1) the keyword RECORD, REALM, SET or 2) keyword SET and appropriate data base names, or contains keywords RETAINING and MULTIPLE.

- Record selection expression - The various record selection expressions are given in Table 3-12 under the explanation of the FIND statement.
- Statement number - consists of one to five digits
- Subroutine name - consists of one to six letters or digits, the first of which must be a letter.
- Usage - one of each of the following groups of keywords: 1) PROTECTED, EXCLUSIVE, CONCURRENT 2) RETRIEVAL, UPDATE.

TABLE 3-6. LIST ITEM SEQUENCE ELEMENTS

- Input list item which must be one of the following: variable name, array element name, character substring name, array name, or array block item (cf. FORTREV 12.8.2.1).
- Implied-DO list item consisting of one of the following: a variable name, an array element name, a character substring name, an array name, or an array block item (cf. FORTREV 12.8.2.3).

3.2 SUBSCHEMA INFORMATION PROCESSING

After the identifier, keyword, and list item sequences have been constructed, the various components of the DML statement containing subschema, REALM, SET, record, privacy, and error information must be extracted from these sequences. This information is then appropriately entered into one of the following tables: subschema, realm, set, record, or error status. The record description for each of these tables is given in Tables 3-7 through 3-11, respectively. These record description tables contain specific information about the subschema which can be collected from the DML statements. Moreover, the printing of these tables in a highly readable format will provide the user with different descriptions of the data base components which were established by the DML statements. With these labels, the user can cross reference the information and thereby determine the consistency of the data base descriptions within the bounds of the applications program.

3.3 FUNCTIONAL REQUIREMENTS FOR THE DBVS

As stated previously in Part I of this report, the specifications for the CODASYL DML are not in final form. The pertinent section of the CODASYL FORTRAN Data Base Facility Journal of Development (November 25, 1975), written by the Data Base Manipulation Language Committee, was presented in the January 1976 monthly progress report in Appendix A. It should be used as a reference for understanding the subsystem software requirements document subsequently presented in Figure 3-1.

TABLE 3-7. SUBSCHEMA TABLE RECORD DESCRIPTION

Item Name	Data Type
Subschema Name	(cf. FORTREV, Section 4.8.1)
Schema Name	(cf. FORTREV, Section 4.8.1)
Privacy Key (assoc. with INVOKE or PRIVACY statement)	(cf. FORTREV, Section 4.8.1)
DML Command Indicator	Integer constant Integer data
Indicates reference by one of the following DML commands: INVOKE PRIVACY	
Line Number	Integer constant Integer data
Indicates line number of DML command in listing	

TABLE 3-8. REALM TABLE RECORD DESCRIPTION

Item Name	Data Type
Realm Name	Integer variable Hollerith data
Set Names	Integer variables Hollerith data
Usage indicator	Integer constant Integer data
Indicates one item from each of the following groups: 1. PROTECTED, EXCLUSIVE, CONCURRENT 2. RETRIEVAL, UPDATE	
Statement number or subroutine name for error handling	Integer variable Hollerith data
Record Name	Integer variables Hollerith data
Subschema name	(cf. FORTREV, Section 4.8.1)
Privacy Key (assoc. with PRIVACY statement)	(cf. FORTREV, Section 6.2)
DML Command Indicator	Integer constant Integer data
Indicates reference by one of the following DML commands: READY, FINISH, FIND, FETCH, OR PRIVACY	
Line Number	Integer constant Integer data
Indicates line number of DML command in listing	

TABLE 3-9. SET TABLE RECORD DESCRIPTION

Item Name	Data Type
Set Name	Integer variable Hollerith data
Record Names	Integer variable Hollerith data
Modification Indicator Indicates that set relationship has been changed within the program.	Integer constant Integer data
Privacy Key (assoc. with PRIVACY statement)	(cf. FORTREV, Section 6.2)
DML Command Indicator Indicates reference by one of the following DML commands: CONNECT, DISCONNECT, MODIFY, FIND, FETCH, ERASE, or PRIVACY	Integer constant Integer data
Statement Number or Subroutine Name for Error Handling	Integer variable Hollerith data
Record Delete Indicator	Integer constant Integer data
Subschema Name	Integer variable Hollerith data
Line Number Indicates line number of DML command in listing.	Integer constant Integer data

TABLE 3-10. RECORD TABLE RECORD DESCRIPTION

Item Name	Data Type
Record Name	Integer variable Hollerith data
Set Name	Integer variables Hollerith data
Realm Name	Integer variables Hollerith data
Subschema Name	(cf. FORTREV, Section 4.8.1)
Modification Indicator Indicates that set relationship has been changed within the program.	Integer constant Integer data
Statement Number or Subroutine Name	Integer variables Hollerith data
Data Base Key Name	Integer variables Integer data
Data Item Names	Integer variables Integer data
Privacy Key (assoc. with PRIVACY statement)	(cf. FORTREV, Section 6.2)
DML Command Indicator Indicates reference by one of the following DML commands: FIND, GET, FETCH, STORE, MODIFY, ERASE, CONNECT, DISCONNECT, or PRIVACY	Integer constant Integer data

TABLE 3-10. (Continued)

Item Name	Data Type
Line Number	Integer constant Integer data
Indicates line number of DML command in listing	
DML Statement Indicator	Integer constant Integer data
Indicates privacy access by one of the following DML commands: INSERT, REMOVE, STORE, MODIFY, FIND, GET, FETCH, or ERASE	
Record Delete Indicator	Integer constant Integer data
Indicates PERMANENT, SELECTIVE, or ALL record delete	

TABLE 3-11. ERROR STATUS TABLE
RECORD DESCRIPTION

Item Name	Data Type
Procedure Name	Integer variable Hollerith data
ALL indicator	Integer variable Hollerith data
OTHER indicator	Integer variable Integer data
STATUS indicator	Integer variable Integer data

Problem Statement:

Design a data base verifier subsystem that analyzes FORTRAN Data Manipulation Language (DML) statements for the purpose of ensuring a consistent and valid data base evocation by the program. Print the results of this analysis in a form easily interpretable by the user.

SYSTEM SOFTWARE REQUIREMENTS DOCUMENT

Direction

Design a data base verifier subsystem that analyzes all FORTRAN DML statements and organizes the subschema, realm, set, record, privacy, and error information contained within the DML statements into appropriate output for the user.

Input

LINE_BUFFER: FORTRAN DML source card

Transductions

INITIALIZE_SYSTEM: Initialize DML command, character, and keyword tables, line number counter, etc.

CHECK_DML_COMMAND: Check first keyword for a match in the DML command (Table 3-3)

CHECK_DML_KEYWORD: Check keyword for a match in the keyword table (Table 3-4)

BUILD_KEYWORD_SEQ: Build keyword sequence for each DML statement

BUILD_IDENTIFIER_SEQ: Build identifier sequence for each statement

Figure 3-1. Subsystem Software Requirement Document for DBVS

BUILD_LIST_ITEM_SEQ: Build list item sequence for each statement

BUILD_DML_TABLE: Construct DML command table

BUILD_KEYWORD_TABLE: Construct keyword table

READ_LINE: Read a line of source code

EVOKE_MODULE: Evoke appropriate module after processing each DML statement or QUIT command.

Output

SUBSCHEMA_INFO: Subschema, realm, set, record, privacy, and error handling information in tabular form

Constraints

Input: The source code must be written in CODASYL extended FORTRAN which contains DML statements for program/data base interaction. The acceptable input formats are listed in Table 3-12.

Implications

BUILD_DML_TABLE $\subset$ CHECK_DML_COMMAND

BUILD_KEYWORD_TABLE $\subset$ CHECK_DML_KEYWORD

READ_LINE $\subset$ CHECK_DML_COMMAND

READ_LINE $\subset$ CHECK_DML_KEYWORD

READ_LINE $\subset$ BUILD_KEYWORD_FILE

Figure 3-1. (Continued)

TABLE 3-12. INPUT FORMATS

```
1.  INVOKE(SUBSCHEMA=<char const>*,SCHEMA=<char const>*
        {, PRIVACY = <char const> *} )

2.  READY({ALL|{SET = <db names>*}|{REALM = <db names>*}},
        <usage>*[,<error>*])

3.  FINISH({ALL|{SET = <db names>*}|{REALM = <db names>*}}
        [,<error*>])

4.  CONNECT([RECORD = <db name>*,]{ALL|SET =<db names>*}},
        [,<error>*])

5.  DISCONNECT([RECORD =<db name>*,]{ALL |{SET =<db names>*}},
        [, <error> *])

6.  ACCEPT(CURRENCY =<db key>[,<currency type> ][,<error>])
        <currency type>::={{RECORD|SET}=<db name>|RUNUNIT|
        REALM=<db name>
        or
        ACCEPT(REALM NAME=<char var>[,<name type>][,<error>])
        <name type>::={{RECORD|SET}=<db name>}| RUNUNIT|{KEY=
        <db key>}

7.  FIND({<rse>|{RSE=<char exp>*}}[,<retain>*][,<error>*])
        <rse>:: =<rse 1>|<rse 2>|<rse 3>|<rse 4>|<rse 5>|
        <rse 6> | <rse 7>
        <rse 1>::= KEY =<db key>[,RECORD=<db name>]
        <rse 2>::={DUPLICATE|ANY},RECORD=<db name>
        <rse 3>::  DUPLICATE,SET=<db name>,USING=<id list>
        <rse 4>::=<posit>[,RECORD=<db name>]
            [,{REALM|SET}=<db name>]
        <posit>::={OFFSET=<int exp>}|FIRST|LAST|NEXT|PRIOR
        <rse 5>::=CURRENT[,RECORD=<db name>][{REALM|SET}=
            <db name>]
        <rse 6>::=OWNER,SET=<db name>
        <rse 7>::=RECORD=<db name>,[CURRENT]SET=<db name>
            [,USING=<id list>]

8.  GET([RECORD=<db name>][,<error>])[<item list>]

9.  FETCH({<rse>|{RSE=<char exp>*}}[,<retain>*][,<error>*])
        [<item list>*]

10. STORE (RECORD=<db name>*[,<retain>*][,<error>*])
```

* Meta symbols are explained in Table 3-13.

TABLE 3-12. (Continued)

```
11.   MODIFY([RECORD=<db name>*]
          ([,ALL|{SET=<db names>*}][,ONLY][,<error>*])
          [<id list>*])

12.   ERASE([RECORD=<db name>*,][<member type>]
          [,MEMBERS=<db names>*][,SET=<db names>*]
          [,<error>*])
      <member type>:: = PERMANENT|SELECTIVE|ALL

13.   PRIVACY(SUBSCHEMA =<char con>*[,DISPLAY],<priv key>*)
      <priv key>::=PRIVACY={<char exp>|<priv key proc>}
      <priv key proc>:: = FORTRAN subroutine name

      PRIVACY[(SUBSCHEMA=<name>*,]REALM=<db name>*
          ,<usage mode>*,<priv key>*)
      <usage mode>::=
          [,{PROTECTED|EXCLUSIVE}][,RETRIEVAL|UPDATE]

      PRIVACY([SUBSCHEMA=<char con>*,]RECORD=<db names>*
          [,<dml stmt>*],<priv key>*)

      <dml stmt>::=

          {INSERT|REMOVE|STORE|<erase>|MODIFY|FIND|GET
           FETCH}

      <erase>::=ERASE{PERMANENT|SELECTIVE|ALL}

      PRIVACY([SUBSCHEMA=<char con>*,]SET=<db names>*
          [,<dml type>*],<priv key>*

      <dml type>::=ORDER|CONNECT|DISCONNECT|<erase>|FIND|
          FETCH

      <erase>::= ERASE{PERMANENT|SELECTIVE|ALL}

      PRIVACY([SUBSCHEMA=<char con>*,]<item dml>,<priv key>*)
          [<id list>*]

      <item dml>::=MODIFY{GET|FETCH}
```

* Meta symbols are explained in Table 3-13.

TABLE 3-12. (Continued)

14. USE(PROCEDURE=<identifier>*[,{ALL|OTHER|{STATUS=
<status list>*}}]

15. ORDER({SET=<db name>*}[,LOCALLY]{,<sort spec>...}[,<error>])

<sort spec>::={ASCENDING|DESCENDING}=(RECORD|KEY|<sort field>...)

<sort field>::=ITEM=<id list>|KEY=<db names>|RECORD=<db names>

* Meta symbols are explained in Table 3-13.

TABLE 3-13. META-SYMBOL MEANINGS

Meta-Symbols	Meaning
<char var>	character variable - FORTREV variable of type character
<char const>	character constant - is an apostrophe followed by a non-empty string of characters followed by an apostrophe
<char exp>	character expression - is used to express a character string consisting of a character primary alone or concatenated with other character primaries. A character primary may be a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, character function reference, or character expression enclosed in parentheses. (See FORTREV 75-09-26, Section 4)
<db key>	data base key - integer variable (FORTREV's definition of variable excludes array elements)
<db name>	data base name - record name, set name, realm name, or character expression. (The first 3 are names in the subschema being used)
<db names>	data base names - data base name
<error>	error phrase - contains keyword ERROR and either statement number or subroutine (with arguments if applicable) name
<id spec>	item specification - input list item which must be one of the following: variable name, array element name, character substring name, array name, or array block item (See FORTREV 12.8.2.1)

TABLE 3-13. (Continued)

Meta-Symbols	Meaning
<identifier>	identifier - item specification
<id list>	identifier list - item specification(s)
<item spec>	identifier specification - item specification or implied-DO lists consisting of one of the following: a variable name, an array element name, a character substring name, an array name or an array block item (12.8.2.3)
<item list>	item list - identifier specification
<priv key proc>	subroutine name - consists of one to six letters or digits, the first of which must be a letter
<retain>	retain - contains keyword RETAINING and either 1) the keyword RECORD, REALM, SET OR 2) keyword SET and appropriate data base names; or contains keywords RETAINING and MULTIPLE
<usage>	usage - one of each of the following groups of keywords: 1) PROTECTED, EXCLUSIVE, CONCURRENT 2) RETRIEVAL, UPDATE
<rse> (See Table 3-13 number 7 input format)	record selection expressions - are used to specify criteria whereby the data base management system is to select a record in the data base. The various record-selection expressions are used as follows:
<rse 1>	Format 1 - direct access
<rse 2>	Format 2 - calculate mode access
<rse 3>	Format 3 - set search access
<rse 4>	Format 4 - positional access
<rse 5>	Format 5 - currency indicator access
<rse 6>	Format 6 - set owner access
<rse 7>	Format 7 - set occurrence selection rules access

3.4 FUNCTIONAL SPECIFICATIONS FOR THE DBVS

To facilitate the understanding of the DBVS functional specifications, the following documents should be used as references:

- the portion of the "CODASYL FORTRAN Data Base Facility Journal of Development"(November 25, 1975) which was printed in Appendix A of the January 1976 monthly progress report
- the special report, "SSL-A Software Specification Language," which was prepared for this contract
- the"draft proposed ANS FORTRAN, BSR X3.9, X3J3/76" which appeared in the March 1976 issue of SIGPLAN Notices. Any references to FORTREV appearing in the specifications are synonymous with those of the aforementioned ANS FORTRAN document.

In the special report on SSL, explicit descriptions of the preamble components as well as subsystem and module descriptions for designing a software system are presented. A simple outline of this document (Table 3-14) will serve as a guide to understanding the SSL description of the data base verifier. In accordance with the SSL report, the subsystem preamble will be presented first (Figure 3-2) and then the module descriptions (Figures 3-3 through 3-10). To accompany these specifications, a module structure chart for the DBVS (cf. Figure 3-11) and a table containing a summary of the module descriptions (cf. Table 3-15) are given.

TABLE 3-14. OUTLINE OF SSL COMPONENTS

I. Preamble Description
    1. Requirement Declaration
        a. Input and Output Parts
        b. Transduction Parts
        c. Constraint Declarations
    2. Data Type and Variable Declarations
        a. Simple Types
        b. Structured Types
        c. Pointer Types
    3. Constant Declarations

II. Module Description
    1. Module Statement
    2. Assumes and Satisfies Statements
    3. Fulfills Statement
    4. Accesses Statement
    5. Receives and Transmits Statements
    6. Creates, Modifies, and Uses Statements
    7. Execute Statement

III. Subsystem Description
    1. Subsystem Preamble
    2. Module Description

```
/*preamble for DBVS subsystem*/

Requirement
        Input line_buffer; /*FORTRAN dml source card*/
        Transductions
            initialize_system; /*initialize dml command, char-
               acter, and keyword tables, line number
               counter,*)
            check_dml_command;/*check first keyword for a
               match in the dml command table (Table 3-3)
               including terminate keyword quit*/
            check_dml_keyword;/*check keyword for a match
               in the keyword table (Table 3-4)*/
            build_keyword_seq;/*build keyword sequence for
               each dml statement*/
            build_identifier_seq;/*build identifier sequence
               for each statement*/
            build_list_item_seq;/*build list item sequence
               for each statement*/
            build_dml_table in check_dml_command;/*construct
               dml command table*/
            build_keyword_table in check_dml_keyword;
               /*construct keyword table*/
            read_line in check_dml_command, check_dml_
               keyword, build_keyword_seq;/*read a line
               of source code*/
            evoke_module;/*evoke appropriate module after
               processing each dml statement or quit
               command*/
```

Figure 3-2. DBVS Subsystem Preamble

```
save_subschema_info;/*save subschema, schema or
    privacy key names, and the dml command
    indicator in subschema table (Table 3-7)*/

save_realm_info;/*store set or realm information
    and usage information in realm table (Table
    3-8)*/
save_set_info;/*store record (if specified) and
    set information in record and set tables,
    (Table 3-9 and 3-10,respectively)
save_record_info;/*store record name information,
    and if specified, item dml, record items, and
    privacy key information for record in record
    table (Table 3-10)and pertinent record infor-
    mation in set table (Table 3-9)*/
save_subschema-privacy in save_subschema_info;
    /*store subschema name and privacy key infor-
    mation for subschema in subschema table*/

save_realm_privacy in save_realm_info;
    /*store realm name, subschema name (if
    specified), usage indicator, and privacy
    key information in realm table*/
save_set_privacy in save_set_info;/*store set
    name, subschema name (if specified) dml
    type indicator, and privacy key information
    for set in set table (dml type indicator
    reflects one of the following commands; order,
    connect, disconnect, find, fetch, permanent,
    selective, or all)*/
```

Figure 3-2. (Continued)

```
            save_record_privacy in save_record_info;/*store
               record name and if specified, subschema name,
               dml statement (indicating insert, remove,
               store, erase (permanent, selective, all),
               modify, find, get, fetch) and privacy key
               information for record in record table*/
            save_item_privacy in save_record_info;/*store
               item dml (modify, get, fetch) and privacy key
               information and if specified, subschema name
               and item names in record table*/
            save_error_table_info;/*store error information
               in error table (Table 3-11 )*/
            print_tables;/*read and write information from
               subschema, realm, set, record, and error
               tables and write to printer*/
         Output subschema_info;/*subschema, realm, set,
            record, privacy, and error handling information
            in tabular form*/
         end;

/* beginning of data description within the preamble*/
         variable dml_intrinsic; array [1..5] of text;/*
            contains current dml command for check_dml_
            command*/
            for check_dml_command, check_dml_keyword, build_
               keyword_seq, build_identifier_seq, build_list_
               item_seq, evoke_module, read_line, save_sub-
               schema_info, save_subschema_privacy, save_
               realm_ info, save_realm_privacy, save_set_
               info, save_set_privacy, save_record_info,
               save_record_privacy, save_item_privacy;
```

Figure 3-2. (Continued)

```
variable keyword_table: array [1..39, 1..5] of
    text;/*Table 3-4(allowable keywords within dml
    statements)*/
    for build_keyword_table, check_dml_command,
        check_dml_keyword, build_keyword_seq, build_
        identifier_seq, build_list_item_seq, evoke_
        module, read_line, save_subschema_info,
        save_subschema_privacy, save_realm_info,
        save_realm_privacy, save_set_info, save_set_
        privacy, save_record_info, save_record_pri-
        vacy,save_item_privacy, save_error_table_
        info, build_dml_table, initialize_ system;
variable list_item: array [1..3] of text;/* con-
    tains user specified list item*/
    for check_dml_command, check_dml_keyword, build_
        keyword_seq, build_identifier_seq, build_
        list_item_seq, evoke_module, read_line,
        save_record_info, save_record_privacy, save_
        item_privacy;
variable identifier; array [1..3] of text;/* con-
    tains user specified identifier*/
    for check_dml_command, check_dml_keyword, build_
        keyword_seq, build_identifier_seq, build_
        list_item_seq, evoke_module, read_line,
        save_subschema info, save_subschema_privacy,
        save_realm_info, save_realm_privacy, save_
        set_info, save_set_privacy, save_record_info,
        save_record_privacy, save_item_privacy, save_
        error_table_info;
type ident_seq=sequence of text
    ident: array [1..3] of text/* identifier name*/
```

Figure 3-2. (Continued)

```
variable dml_command_table: array [1..16, 1..5] of
    text;/*contains commands of Table 3-3*/
    for build_dml_table, check_dml_command, check_dml
        keyword, build_keyword_seq, build_identifier_
        seq, build_list_item_seq, evoke_module, read_
        line, initialize_system, build_keyword_
        table;
variable keyword: array [1..5] of text;/* contains
    user specified keyword*/
    for check_dml_command, check_dml_keyword, build_
        keyword_seq, build_identifier_seq, build_
        list-item_seq, evoke_module, read_line, save_
        subschema_info, save_subschema_privacy, save_
        realm_info, save_realm_privacy, save_set_
        info, save_set_privacy, save_record_privacy,
        save_item_privacy, save_error_table_info,
        save_record_info;
type line_buffr = array [1..72] of char;
        /*reflection of program card*/
variable line_buffer:  line_buffr;/*contains current
    source statement being analyzed*/
    for read_line, check_dml_command, check_dml_
        keyword, build_keyword_seq, build_identifier_
        seq, build_list_item_seq, evoke_module;
variable char_table; array [1..49] of char;/*
    A..Z,  0..9  , blank, =, +, *, /, (, ), ,, ., ;,
    :, $*/:
    for check_dml_command, check_dml_keyword, build_
        keyword_seq, build_identifier_seq, build_
        list_item_seq, evoke_module, read_line,
        initialize_system, build_dml_table, build_
        keyword_table;
```

Figure 3-2. (Continued)

```
variable identifier_seq: ident_seq;/*sequence
    containing all identifiers associated with
    one statement*/
    for build_identifier_seq,check_dml_command,
        check_dml_keyword, build_keyword_seq, build_
        list_seq, evoke_module, read_line, save_
        subschema_info, save_subschema_privacy, save_
        realm_info, save_realm_privacy, save_set_
        info, save_set_privacy, save_record_info,
        save_record_privacy, save_item_privacy, save_
        error_table_info;
type list_itm_seq = sequence of text
    parametr: array [1..3] of text;/*list_item_name*/
    end;
variable list_item_seq: list_itm_seq;/*sequence
    containing all list items associated with one
    statement*/
    for build_list_item_seq, check_dml_command,
        check_dml_keyword, build_keyword_seq, build_
        identifier_seq, evoke_module, read_line;
type keywrd_seq = sequence of records
    keyword: array [1..3] of text;/*name of keyword*/
    identifier_counter; integer;/*contains number
        of identifiers (i.e., no. of times to read
        the identifier sequence associated with each
        keyword*/
    end;
variable keyword_seq: keywrd_seq;/*contains keyword
    number of identifier, associated with each
    keyword*/
    for check_dml_command, check_dml_keyword, build_
        keyword_seq, build_identifier_seq, build_
        list_item_seq, evoke_module, read_line,
        build_keyword_ seq, save_subschema_info,
        save_subschema_privacy, save_realm_info,
```

Figure 3-2. (Continued)

```
            save_realm_privacy, save_set_info, save_
            set_privacy, save_record_info, save_record_
            privacy, save_item_privacy, save_error_
            table_info;
        variable list_item_seq_ind:  integer;/*indicates
            number of list items associated with a fetch,
            get, modify, or privacy statement, i.e., it
            indicates number of entries in the list item
            sequence*/
            for check_dml_command, check_dml_keyword, build_
                item_seq, evoke_module, read_line, save_
                record_info, save_record_privacy, save_
                item_privacy, initialize_system, build_dml_
                table, build_keyword_table/;
        type subschema_tabl=file of records
            subschema_name: array [1..3] of text;
            schema_name: array [1..3] of text;
            privacy_key: array [1..3] of text;
            dml_intrinsic: array [1..5] of text;
                /*dml intrinsic represents either the invoke
                or privacy statement*/
            line_no; integer; /*line number of dml command*/

            end;
        variable subschema_table: subschema_tabl;/*the sub-
                schema table contains the subschema name and
                depending on the options exercised by the
                invoke and privacy statements, the schema
                name, and privacy key*/
            for save_subschema_info, save_subschema_privacy,
                print_tables;
        type realm_tabl = file of records
            realm_name: array [1..3] of text;
            set_name: array [1..3] of text;
            usage_indicator: integer;/*indicates either
                protected, exclusive, concurrent and either
                retrieval or update
```

Figure 3-2. (Continued)

```
        error_handling: array [1..3] of text;
        record_name: array [1..3] of text;
        subschema_name; array [1..3] of text;
        privacy_key: array [1..3] of text;
        dml_intrinsic: array [1..5] of text;
            /*dml intrinsic represents one of the
            following statements-ready, finish, find,
            fetch, or privacy*/
        line_no; integer;/*line number of dml command
            in listing*/
        end;
    variable realm_table: realm_tabl;
        for save_realm_info, save_realm_privacy,
            save_record_info, save_record_privacy,
            save_item_privacy, print_tables;
    type set_tabl=file of records
        set_name: array [1..3] of text;
        record_name: array [1..3] of text;
        mod_indicator: integer;/*indicates that set
            relationship has been changed within the
            program*/
        privacy_key: array [1..3] of text;
        dml_intrinsic: array [1..5] of text;/*
            dml intrinsic represents one of the follow-
            ing statements-connect, disconnect, modify,
            find, fetch, erase, or privacy*/
        error_handling: array [1..3] of text;
        record_delete_ind: integer;/*0-indicates no
            record deletions, 1-indicates normal dele-
            tion, 2-indicates permanent deletion 3-
            indicates selective deletion, 4-indicates
            all deletion*/
        subschema_name: array [1..3] of text;
        line_no: integer;/*line number of dml command in
            listing*/
        end;
```

Figure 3-2. (Continued)

```
variable set_table: set_tabl;/* the set table
    contains mainly set and record information
    depending on the options exercised by the
    connect, disconnect, modify, find, fetch, erase,
    or privacy statements*/
    for save_set_info, save_record_info, save_set_
        privacy, save_record_privacy, save_item_
        privacy, print_tables;
type record_tabl=file of records
    record_name:  array [1..3] of text;
    set_name:  array [1..3] of text;
    realm_name:  array [1..3] of text;
    subschema_name:  array [1..3] of text;
    modification_ind:  integer;/* indicates that
        set relationship has been changed within
        the program*/
    error_handling:  array [1..3] of text;
    data_base_key_name:  array [1..3] of text;
    data_item_name:  array [1..5, 1..3] of text;/*
        contains first five names in a list of
        data item names*/
    privacy_key:  array [1..3] of text;
    dml_intrinsic:  array [1..5] of text;/*
        dml intrinsic represents one of the
        following statements:  find, get, fetch,
        store, modify, erase, connect, disconnect,
        or privacy*/
    line_no:  integer;/*line number of dml command
        in listing*/
    dml_stmt_ind:  integer;/*indicates privacy access
        by one of the following dml commands: insert,
        remove, store, modify, find, get, fetch, or
        erase*/
    record_delete_ind: integer;/*indicates that
        record is to be deleted*/
    end;
```

Figure 3-2. (Continued)

```
variable record_table:  record_tabl;/* the record
    table contains primarily record and set informa-
    tion depending on the options exercised by the
    find, get, fetch, store, modify, erase, connect,
    disconnect, or privacy statements*/
    for save_record_info, save_set_info, save_set_
        privacy, save_record_privacy, save_item_
        privacy, print_tables;
type error_status_tabl=file of records
    procedure_name:  array [1..3] of text;
        /* subroutine procedure which is to be
        called in the event a data base exception
        condition is encountered*/
    all-indicator:  integer; /* the indicated pro-
        cedure will be called for all data base
        exception processing*/
    other_indicator:  integer; /*the indicated pro-
        cedure will be called for any data base
        exception conditions which are not previous-
        ly established*/
    status-indicator:  integer;/*the indicated pro-
        cedure will be called whenever any of the
        conditions in the associated status list are
        called*/
    line_no: integer; /*line number of dml command
        in listing */
    end;
variable error_status_table; error_status_tabl; /*
    the error status table contains information con-
    cerning the options exercised in the use state-
    ment*/
    for save_error_table_info, print_tables;
variable line-no-counter: integer; /* contains line
        number of statement as it appears in the
        listing */
```

Figure 3-2. (Continued)

```
            for check_dml_command, check_dml_keyword,
                build_keyword_seq, build_identifier_seq.
                build_list_item_seq, evoke_module, read_
                line, save_subschema_info, save_subschema_
                privacy, save_realm_info, save_realm_
                privacy, save_set_info, save_set_privacy,
                save_record_info, save_record_privacy, save_
                item_privacy, save_error_table_info,
                initialize_system, build_dml_table, build_
                keyword_table;

end;

/*end of data description within the preamble*/
/*end of preamble for DBVS subsystem*/
```

```
/*module description for dml recognizer and control module*/
module dml_recognizer;
        fulfills check_dml_command, check_dml_keyword,
           build_keyword_seq, build_identifier_seq,
           build_list_item_seq, evoke_module, read_line;

           /*initialize system*/
        creates line_buffer, dml_intrinsic, keyword, list_
           item, identifier, identifier_seq, list_item_seq,
           keyword_seq;

           /*read card and fill line buffer*/
        accesses card_reader; modify line_buffer;

           /*determine dml command*/
        modifies dml_intrinsic using line_buffer, char_table,
           dml_command_table;

           /*isolate dml keyword*/
        modifies keyword using line_buffer, char_table,
           keyword table;

           /*construct keyword sequence*/
        modifies keyword_seq using keyword;

           /*isolate identifier*/
        modifies identifier using line_buffer, char_table;

           /*construct identifier sequence*/
        modifies identifier_seq using identifier;

           /*isolate list items for dml statements*/
        modifies list_item using line_buffer, char_table;

           /*construct list item sequence*/
        modifies list_item_seq using list_item;
```

Figure 3-3. DML - RECOGNIZER Module Description

```
        modifies list_item_seq_ind;

        executes conditionally
            subschema_process, realm_process,
            set_process, record_process, error_process, output_
            summary;
        executes initialize_system;

end;
```

Figure 3-3. DML-RECOGNIZER Module Description (Continued)

```
/*module description for initialize system*/
module initialize_system;
        fulfills initialize_system, build_dml_table,
            read_line, build_keyword_table;

        create dml_command_table, char_table, keyword_table,
            line_no_counter, list_item_seq_ind;

            /*construct dml command tables*/
        modifies dml_command_table;

            /*construct char_table*/
        modifies char_table;

            /*construct keyword_table*/
        modifies keyword_table;

            /*initialize line number counter*/
        modifies line_no_counter;

            /*initialize list item sequence indicator*/
        modifies list_item_seq_ind;

end;
```

Figure 3-4. INITIALIZE-SYSTEM Module Description

```
/*module description for subschema processing*/
module subschema_process;

        fulfills save_subschema_info, save_subschema_ privacy;

        creates subschema_table;

            /*retrieve subschema, schema, privacy, or dis-
            play keywords*/
        modifies keyword using keyword_seq @.keyword, keyword
                  table;

            /*retrieve subschema, schema or privacy key
            name*/
        modifies identifier using keyword_seq @.identifier_
                  counter, identifier_seq;

            /*construct record for subschema table*/
        modifies subschema_table @.subschema_name,
                 subschema_table @.schema_name,
                 subschema_table @.privacy_key,
                 subschema_table @.dml_intrinsic,
                 subschema_table @.line_no using keyword, iden-
                   tifier, dml_intrinsic, line_no_counter;

end;
```

Figure 3-5. SUBSCHEMA-PROCESS Module Description

```
/*module description for realm processing*/
module realm_process;

        fulfills save_realm_info, save_realm_privacy;

        creates realm_table;

            /*retrieve all, set, realm, protected, concurrent,
              exclusive, retrieval, update, error keywords*/
        modifies keyword using keyword-seq @.keyword, keyword_
                 table;

            /*retrieve realm, set, subschema, error or
              privacy key name*/
        modifies identifier using keyword-seq @.identifier_
                 counter, identifier_seq;

            /*construct partial record for realm table*/
        modifies realm_table @.realm_name,
                 realm_table @.set_name,
                 realm_table @.usage_indicator,
                 realm_table @.error_handling,
                 realm_table @.subschema_name,
                 realm_table @.privacy_key,
                 realm_table @.dml_intrinsic,
                 realm_table @.line-no, using keyword, iden-
                   tifier, dml_intrinsic, line_no_counter;

end;
```

Figure 3-6. REALM-PROCESS Module Description

```
/*module description for set processing*/
module set-process;

        fulfills save_set_info, save_set_privacy;

        creates set_table, record_table;

            /*retrieves record, all, set, error, only, sub-
              schema, order, connect, disconnect, permanent,
              selective, all, find, and fetch keywords*/
        modifies keyword using keyword_seq @.keyword, keyword_
                 table;

            /*retrieve  record, set, error, privacy, key, or
              subschema name*/
        modifies identifier using keyword_seq @.identifier_
                 counter, identifier_seq;

            /*construct partial record for set table*/
        modifies set_table @.set_name,
                 set_table @.record_name,
                 set_table @.mod_indicator,
                 set_table @.privacy_key,
                 set_table @.dml_intrinsic,
                 set_table @.line-no,
                 set_table @.error_handling,
                 set_table @.subschema_name using keyword,
                   identifier, dml_intrinsic, line_no_counter;

            /*construct partial record for record table*/
        modifies record_table @.record_name,
                 record_table @.set_name,
                 record_table @.modification_ind,
                 record_table @.dml_intrinsic,
                 record_table @.line_no using keyword, iden-
                   tifier, dml_intrinsic, line_no_counter;
```

Figure 3-7. SET-PROCESS Module Description

```
/*module description for record processing*/
module record_process;

        fulfills save_record_info, save_record_privacy,
                 save_item_privacy;

        creates record_table, set_table, realm_table;

            /*retrieve record, key, duplicate, any, set,
              using, first, last, next, prior, realm, offset,
              current, owner, rse, multiple, error, all, only,
              members, permanent, selective, all, subschema,
              insert, remove, store, modify, find, get, or
              fetch keywords*/
        modifies keyword using keyword_seq @.keyword, keyword_
                 table;

            /*retrieve record, key, set, using, offset,
              realm, rse, error, member or subschema name*/
        modifies identifier using keyword_seq @.identifier_
                 counter, identifier_seq;

            /*construct partial record for record table*/
        modifies record_table @.record_name,
                 record_table @.set_name,
                 record_table @.realm_name,
                 record_table @.subschema_name,
                 record_table @.modification_ind,
                 record_table @.error_handling,
                 record_table @.database_key_name,
                 record_table @.data_item_name,
                 record_table @.privacy_key,
                 record_table @.dml_intrinsic,
                 record_table @.line_no,
```

Figure 3-8 RECORD-PROCESS Module Description

```
                record_table @.dml_stmt_ind,
                record_table @.record_delete_ing using
                  keyword, identifier, dml_intrinsic, line_
                  no_counter,  list_item;

            /*construct partial record for set table*/
        modifies set_table @.set_name,
                set_table @.record_name,
                set_table @.mod_indicator,
                set_table @.dml_intrinsic,
                set_table @.line_no,
                set_table @.record_delete_ind using keyword,
                  identifier, dml_intrinsic, line_no_counter;

            /*construct partial record for realm table*/
        modifies realm_table @.realm_name,
                realm_table @.record_name,
                realm_table @.subschema_name,
                realm_table @.dml_intrinsic,
                realm_table @.line_no using keyword, identi-
                  fier, dml_intrinsic, line_no_counter;

            /*retrieve list items (actually data item names)
              that have been saved as a result of a fetch,
              get, modify, or privacy statement*/
        modifies list_item_seq_ind;

end;
```

Figure 3-8. (Continued)

```
/*module description for error processing*/
module error_process;

        fulfills save_error_table_info;

        creates error_status_table;

            /*retrieve procedure, all, other, status keywords*/
        modifies keyword using keyword_seq @.keyword, keyword_
                 table,

            /*retrieve procedure or status names*/
        modifies identifier using keyword_seq @.identifier_
                 counter, identifier_seq;

            /*construct partial record for error status table*/
        modifies error_status_table @.procedure_name,
                 error_status_table @.all_indicator,
                 error_status_table @.other_indicator,
                 error_status_table @.status_indicator,
                 error_status_table @.line_no using keyword,
                   identifier, line_no_counter;

end;
```

Figure 3-9. ERROR-PROCESS Module Description

```
/*module description for print process*/
module output_summary;

        fulfills print-tables:

            /*read and print subschema table records*/
        access line-printer using subschema_table;

            /*read and print realm table records*/
        access line_printer using realm_table;

            /*read and print set table records*/
        access line_printer using set table;

            /*read and print record table records*/
        access line_printer using record_table,

            /*read and print error status table records*/
        access line_printer using error_status_table,

end;
/*end of main subsystem*/
```

Figure 3-10. OUTPUT-SUMMARY Module Description

Table 3-15. Module Descriptions for the DBVS

DML_RECOGNIZER

To isolate data manipulation language (DML) keywords and associated identifiers and list items. To construct appropriate keyword, identifier, and list item sequences. To evoke initialization or report modules, or to evoke appropriate DML statement processor modules.

INITIALIZE_SYSTEM

To initialize the line number counter, the list item sequence indicator, and the following tables: dml_command, character, and keyword.

SUBSCHEMA_PROCESS

To collect subschema information and enter it into the subschema table.

REALM_PROCESS

To collect realm information and enter it into the realm table.

SET_PROCESS

To collect set information and enter it into the set table.

RECORD_PROCESS

To collect record information and enter it into the record table.

Table 3-15. (Continued)

ERROR_PROCESS

To collect error information and enter it into the error status table.

OUTPUT_SUMMARY

To read and print the following tables: subschema, realm, set, record, and error status.

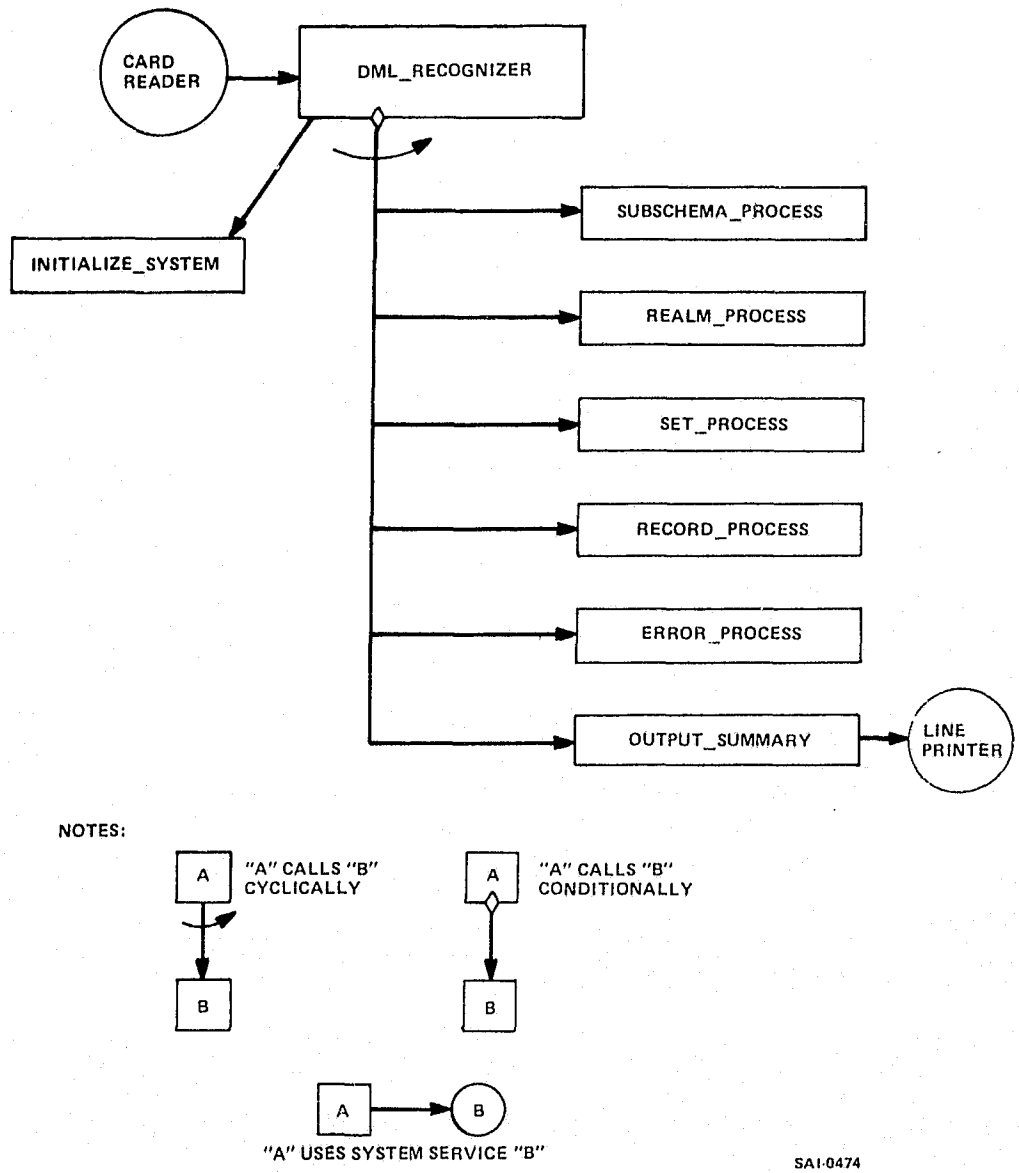


Figure 3-11. Module Structure Chart for DBVS

4. STATIC CODE ANALYSIS

In this section, we present the detailed design (build-to) specifications for the capabilities listed in Table 4-1 which will be incorporated into FACES. For each of the twelve capabilities, we provide a detailed Unit Module Description including a flowchart which is sufficient for coding. However, for complete understanding of this documentation, the following FACES documentation must be used:

- Version 2, Mod 0, Fortran Automatic Code Evaluation System SYSTEM DOCUMENTATION, September 1975, Browne and Ramamoorthy Inc.

- Version 2, FACES User's Manual, September 1975, Browne and Ramamoorthy, Inc.

- Version 2, Mod X FACES Program Listing

The unit module descriptions correspond to those set forth in NASA's "Guidelines For Software Detailed Design Specification (CODE-TO)," while the flowcharts follow ANSI FORTRAN flowcharting recommendations. The detailed specifications for the capabilities will appear in order according to Table 4-1. (Now capabilities 5 and 6 and capabilities 7 and 8 are treated under the same unit module description.

TABLE 4-1. NEW FACES CAPABILITIES

1. EQUIVALENCE and EXTERNAL statements are flagged.

2. COMMONs not named are flagged.

3. ALL COMMON BLOCK arrays must be dimensioned in COMMON BLOCK statements.

4. DIMENSION statement and variable which contain an adjustable (variable) dimension are flagged.

5. Constants, hollerith, or arithmetic expression arguments used in subroutine argument lists are flagged.

6. All occurrences where the same variable exists in multiple positions in an actual parameter list are flagged.

7. Arithmetic IFs are flagged.

8. Targets of branches should not be other branches, especially single GO TOs.

9. Variable which is I/O unit designator is flagged.

TABLE 4-1. NEW FACES CAPABILITIES (Cont.)

10. Statement labels must appear in increasing order.

11. Occurrences of error-prone FORTRAN statements such as ASSIGN statement, assigned GO TO, and PAUSE are flagged.

12. The appearance of the same COMMON variable in more than one DATA statement is flagged.

UNIT MODULE DESCRIPTION

IDENTIFICATION

AIR Modifications

STORAGE ALLOCATION

1K hexadecimal bytes

PURPOSE

Process new query numbers for new FACES capabilities.

DESCRIPTION

These modifications cause AIR to recognize the new query numbers and call the new subroutine or modified subroutine as it processes original query numbers.

HOW ENTERED

Called by LNKAIR

CALLING SEQUENCE

CALL AIR

UNIT MODULE OR OTHER ROUTINES CALLED

New Modules: ER230, ER240, ER250, ER260, ER275, ER280, ER290

Modified Modules: CONALC, COMBAL, MULBRA

SET/USE PARAMETERS

No new parameters are introduced NUMBER, the standard variable for query number is used.

SIGNIFICANT INTERNAL VARIABLES

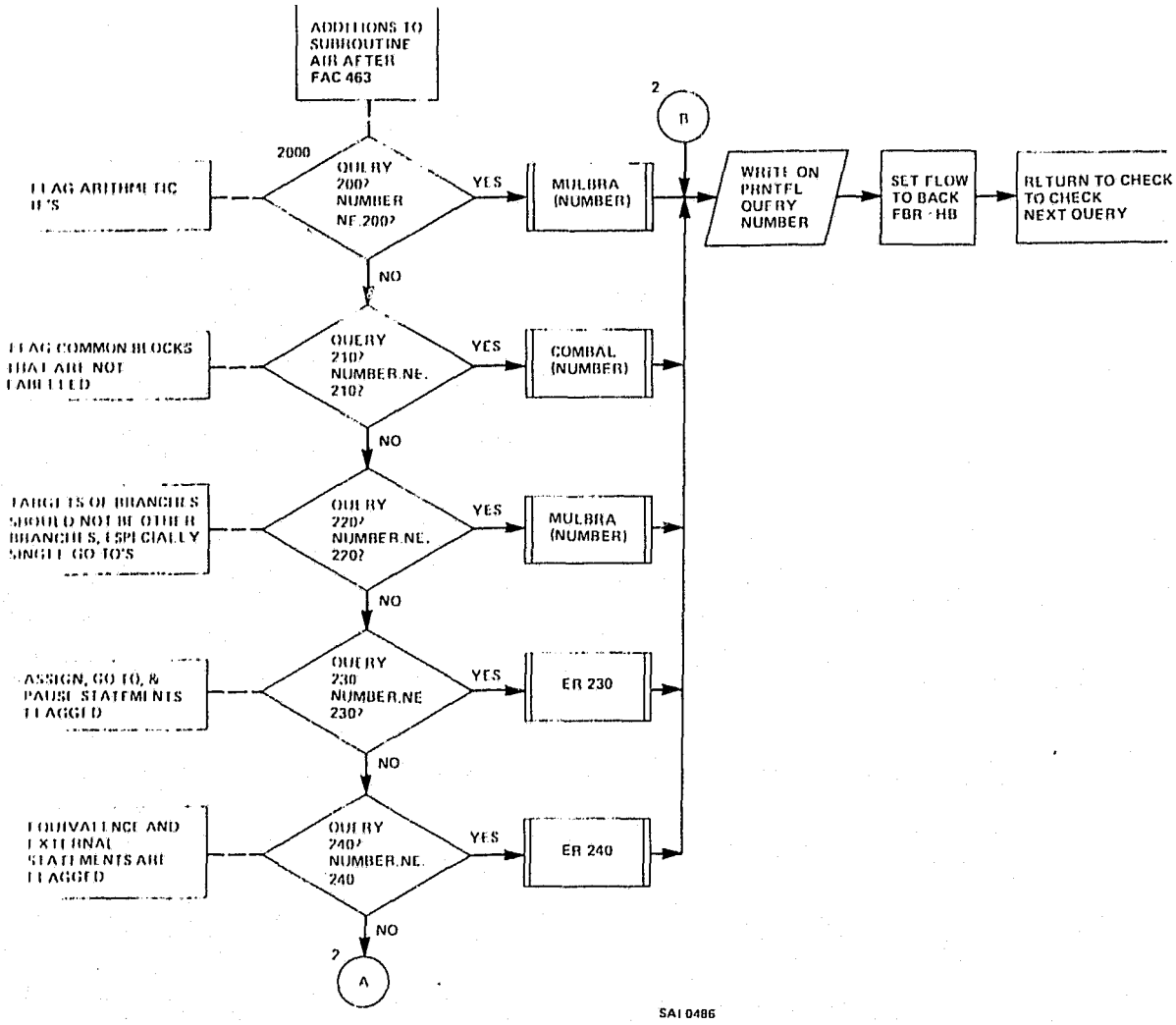
No new variables are introduced.

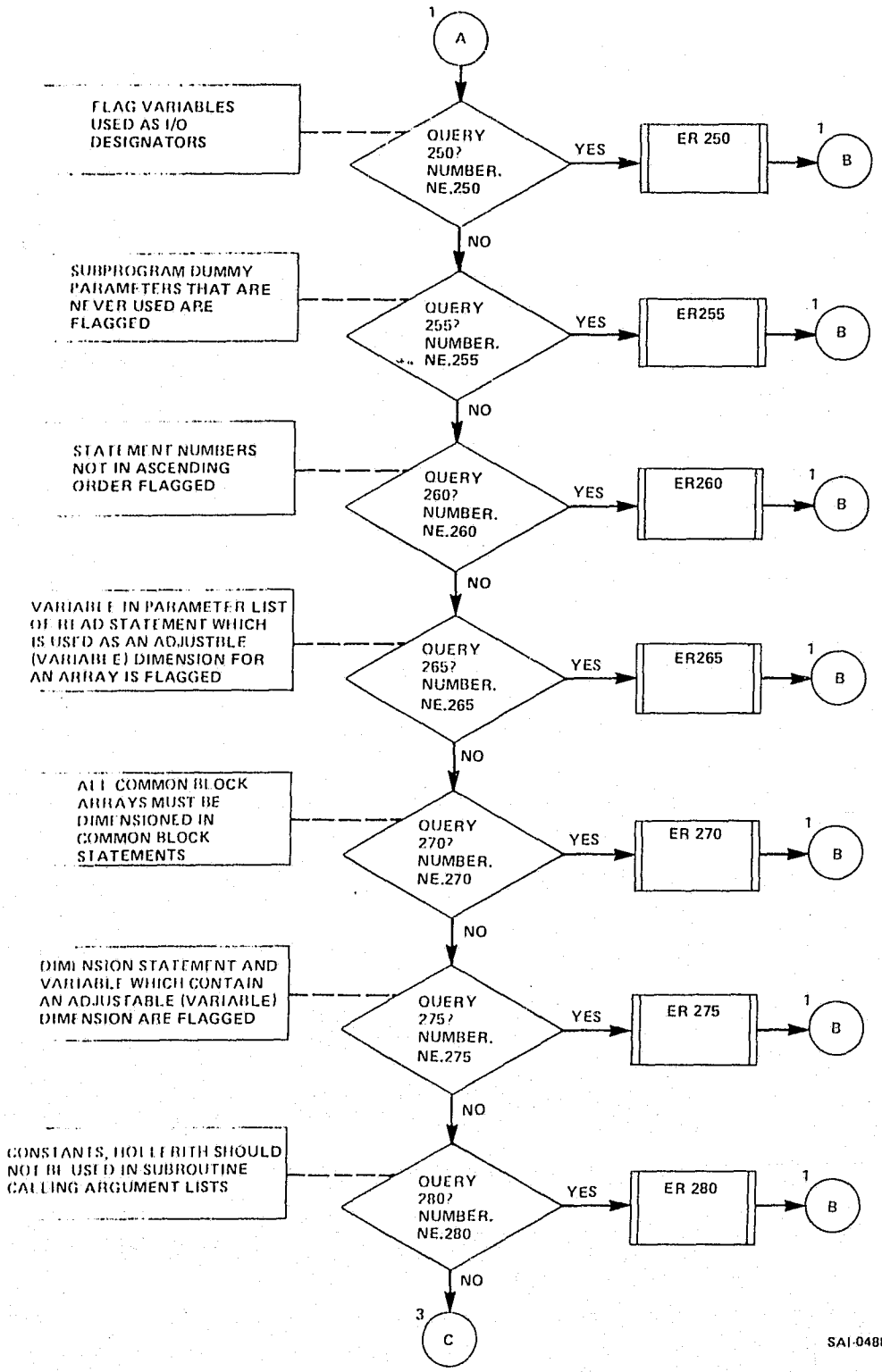
LIMITATIONS AND RESTRICTIONS

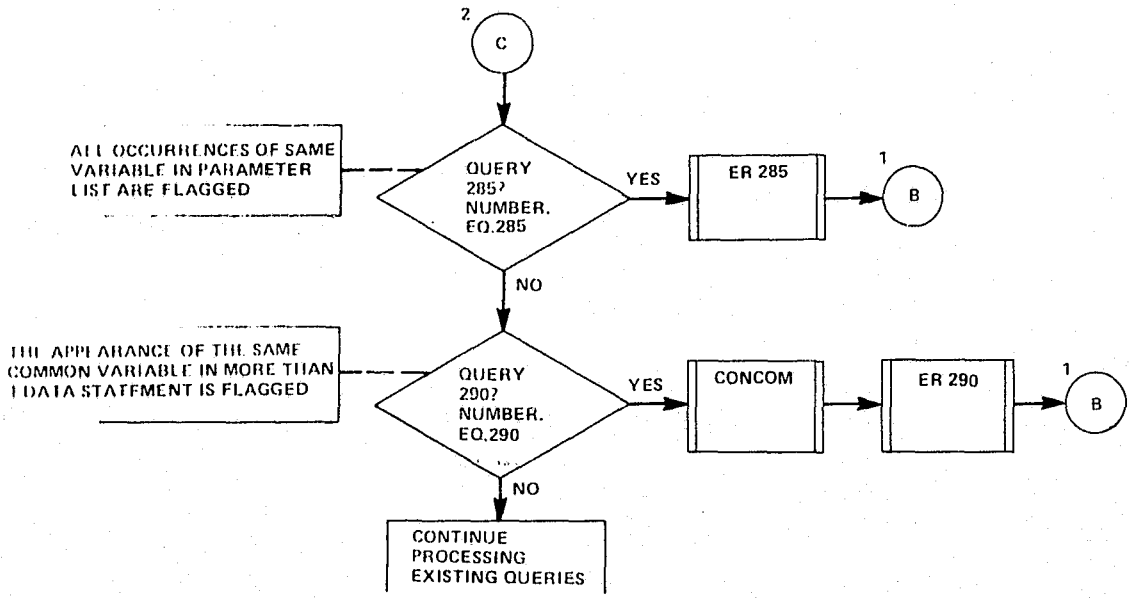
All variables are set by IMPLICIT INTEGER (A-Z).

DETAILED FLOWCHART

Only the modifications are included on the following flowchart.







SAI-0489

UNIT MODULE DESCRIPTION

IDENTIFICATION

ER240 -- Error 240 routine

STORAGE ALLOCATION REQUIREMENT (estimate)

2K (hexadecimal bytes)

PURPOSE

Flag EQUIVALENCE and EXTERNAL statements

DESCRIPTION

The routine checks each module for statement type. If type is a 12 (EQUIVALENCE) or 40 (EXTERNAL) the routine flags the statement in Flag File.

HOW ENTERED

Called by AIR

CALLING SEQUENCE

Call ER240
No Arguments

ROUTINES CALLED

IE
GETE
GETL
POP

SET/USE PARAMETERS

SET

Global: COMMON/SPEREG/

ER(10) -- Error registers

NOTE: Only ER(3) and ER(4) are set.

FBR -- forward/backward register

USE

Global: COMMON/FLAG/

FLAGFL -- input/output designator

COMMON/H/

HB -- hollerith B

HF -- hollerith F

COMMON/TABLE/

HDIR -- directory table

HUSE1 -- use table

HSYM -- symbol table

HNOD -- node table

SIGNIFICANT INTERNAL VARIABLES

I0, I1, I3, I4 -- integer constants

K -- error flag

SCIND -- source code indicator

FSTAT -- first statement number

LSTAT -- last statement number

BF -- general purpose flag

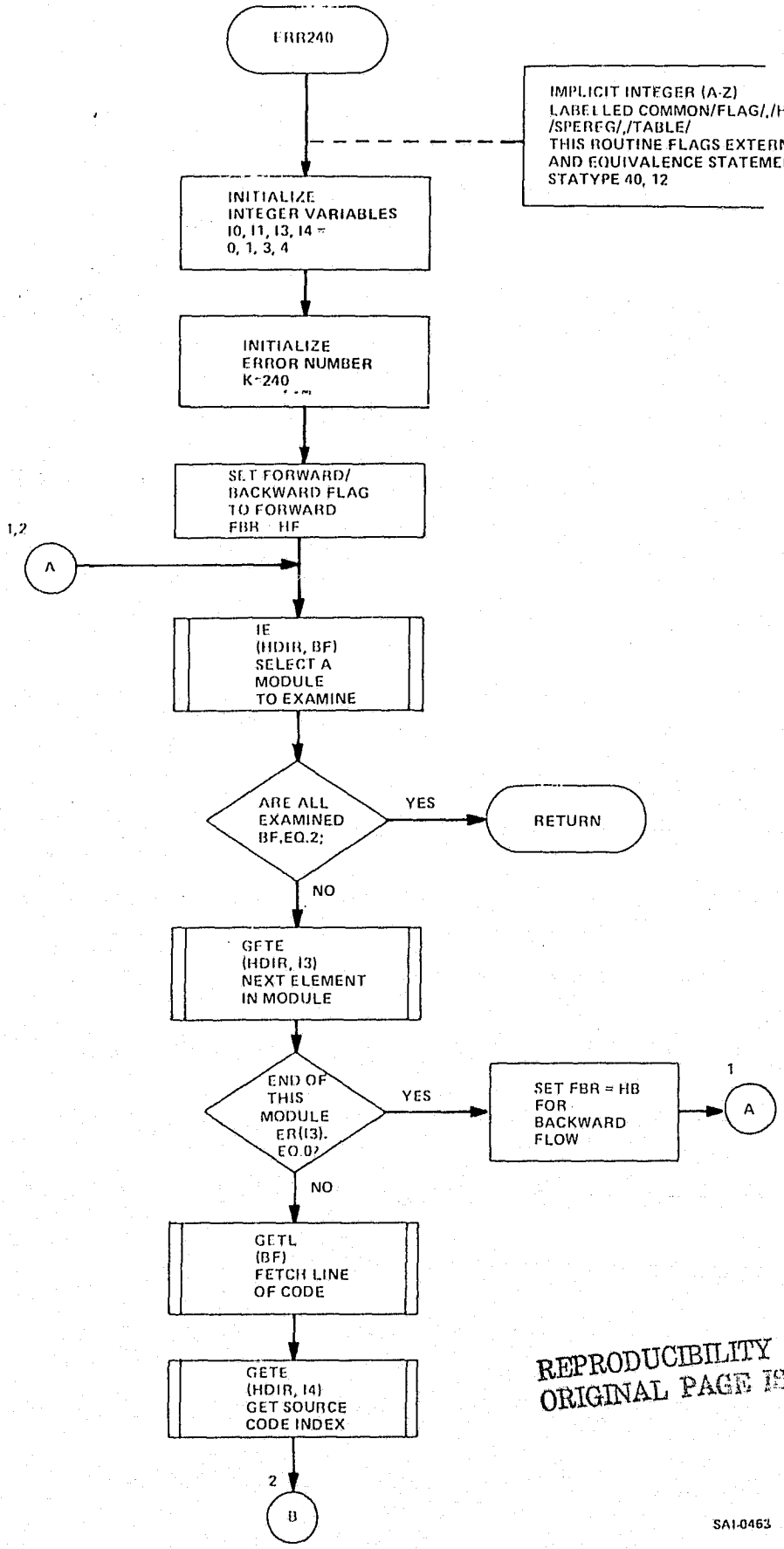
STATYP -- statement type storage

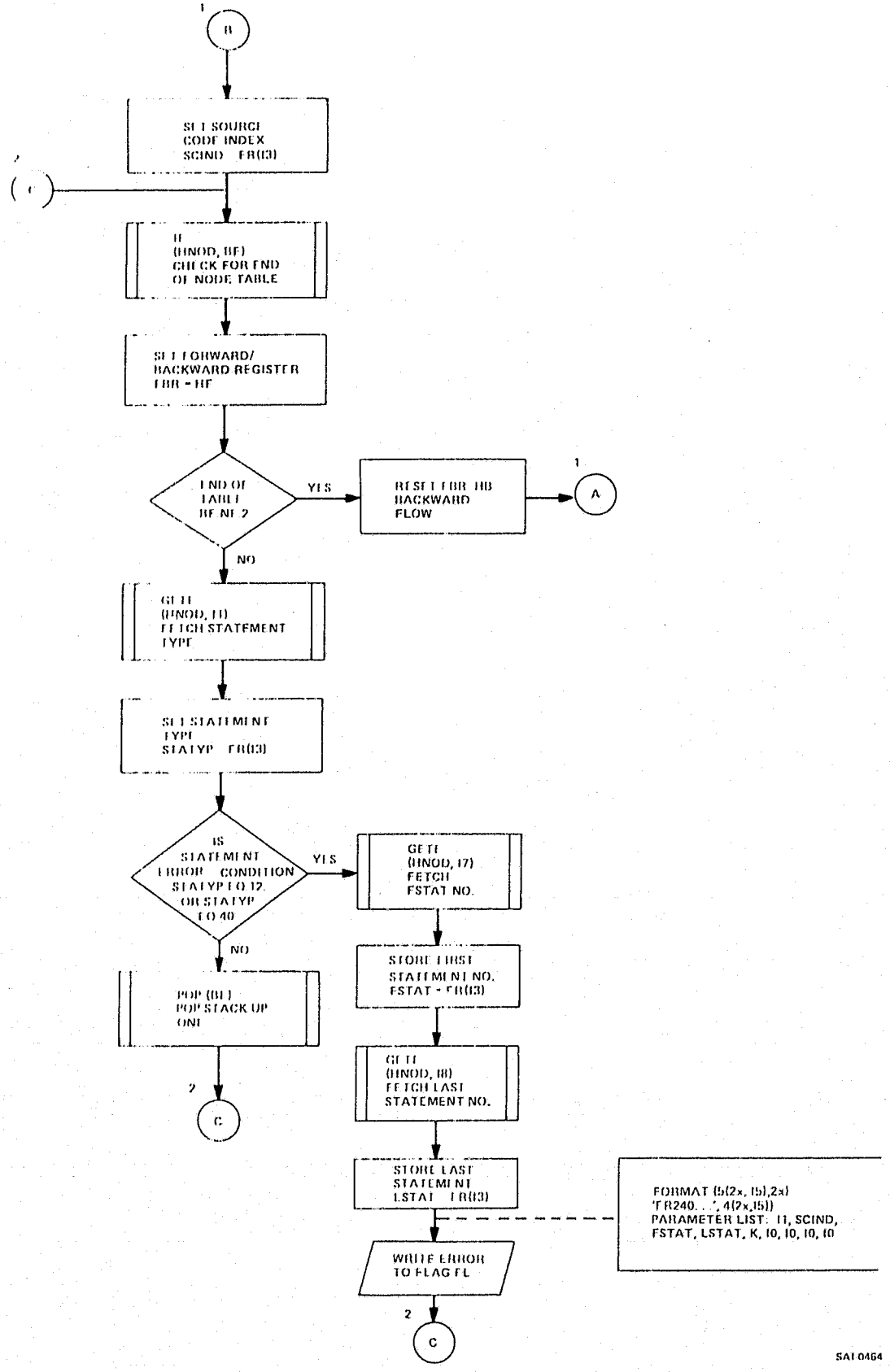
LIMITATIONS AND RESTRICTIONS

All variables are set IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

Attached





UNIT MODULE DESCRIPTION

IDENTIFICATION

COMBAL addition

STORAGE ALLOCATION (estimate)

Additional 250 hexadecimal bytes

PURPOSE

Flag unlabelled COMMON

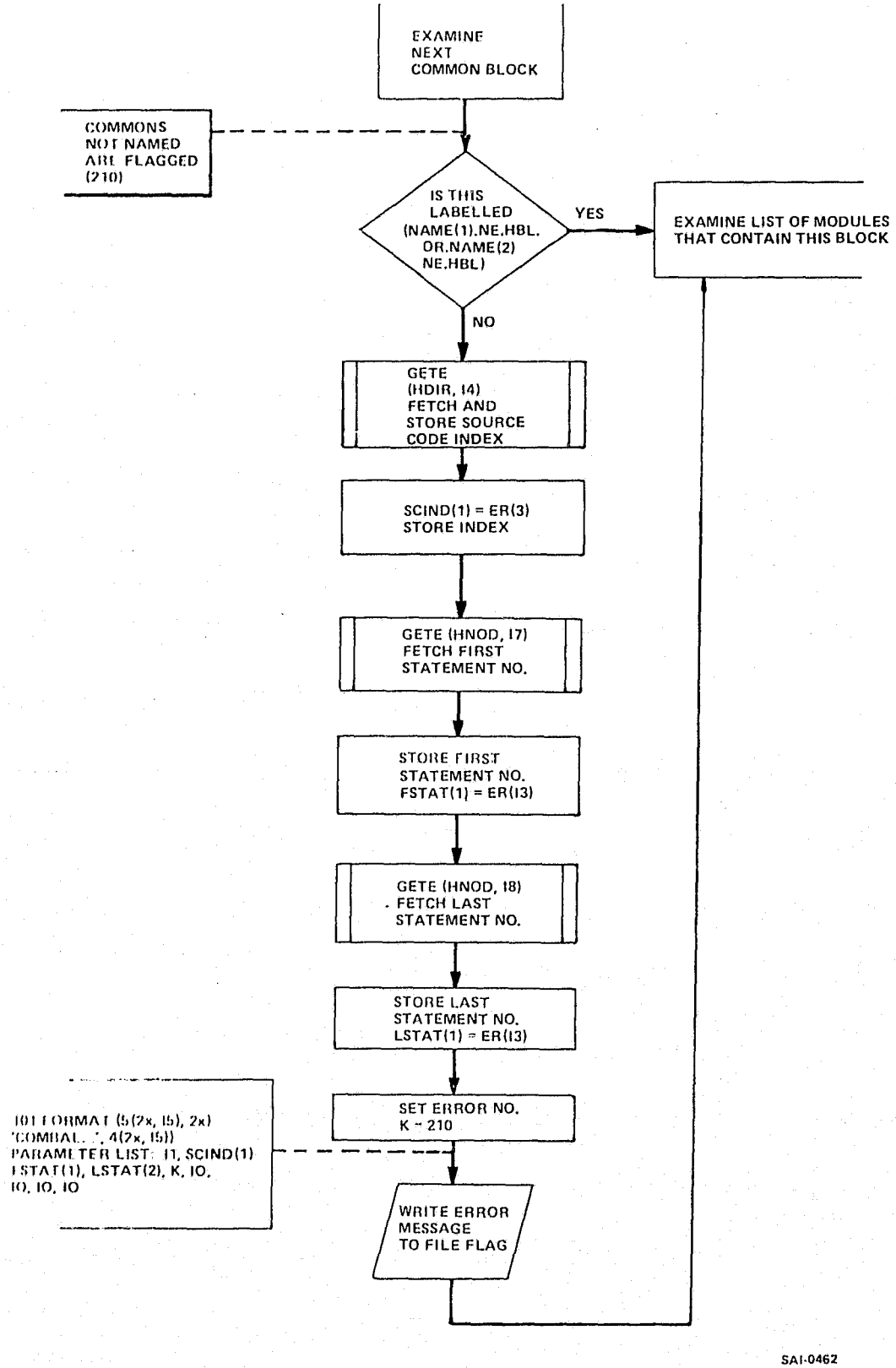
DESCRIPTION

The routine contains a check on the Common Block name. If it is blank the statement is flagged.

NOTE: There are no new variables and subroutines; there is only a new error message and an additional check.

DETAILED FLOWCHART

See attachments



SAI-0462

UNIT MODULE DESCRIPTION

IDENTIFICATION

Modification to CONALC

STORAGE ALLOCATION (estimate)

Additional 500 hexadecimal bytes.

PURPOSE

Flag Common Block arrays that are not dimensioned in Common Block statements.

DESCRIPTION

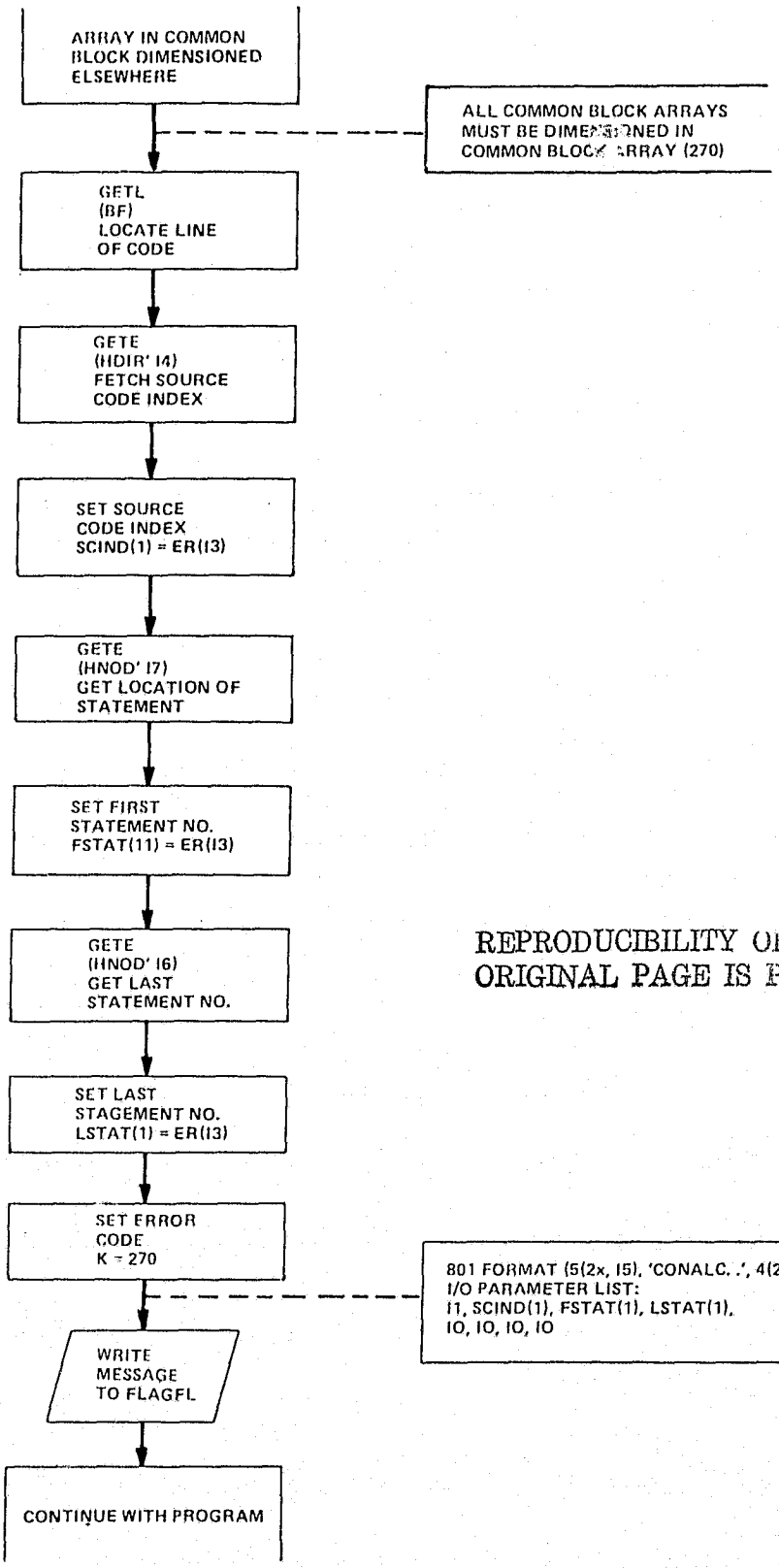
The modification locates the source and writes a message to FLAG FILE to indicate that an array in a COMMON BLOCK is dimensioned elsewhere.

HOW ENTERED

The code occurs at the point where a Common Block variable is dimensioned other than in a Common Block.

DETAILED FLOWCHART

See attachments.



UNIT MODULE DESCRIPTION

IDENTIFICATION

ER275

STORAGE ALLOCATION (estimate)

2K (hexadecimal bytes)

PURPOSE

Flag DIMENSION statement and variable which contains an adjustable (variable) dimension .

DESCRIPTION

This subroutine searches a module for statement type 28 (DIMENSION) then examines each array element for a use code 15 (Array dimension). When this condition exists the error is recorded. At the end of the statement search each array is flagged that has a variable dimension.

HOW ENTERED

Called by AIR subroutine if query is selected. There are no arguments.

CALLING SEQUENCE

Call ER275

OTHER ROUTINES CALLED

IE
GETE
GETL
TT

SET/USE PARAMETERS

USE

COMMON/FLAG/

FLAGFL -- I/O designator

COMMON/H/

HB -- hollerith B

HF -- hollerith F

COMMON/TABLE/

HDIR -- directory

HUSE2 -- linked Test Statement

HUSE1 -- linked List Use

HNOD -- Node Table

COMMON/SPEREG/

ER(10) -- table information turn array

FBR -- program flow register

SIGNIFICANT INTERNAL VARIABLES

I0, I1, I2, I3, I4, I5, I6, I7, I8, I9 = 0 - 9

K -- error flag number

SCIND -- module indicator

FSTAT -- first statement number of error

LSTAT -- last statement number of error

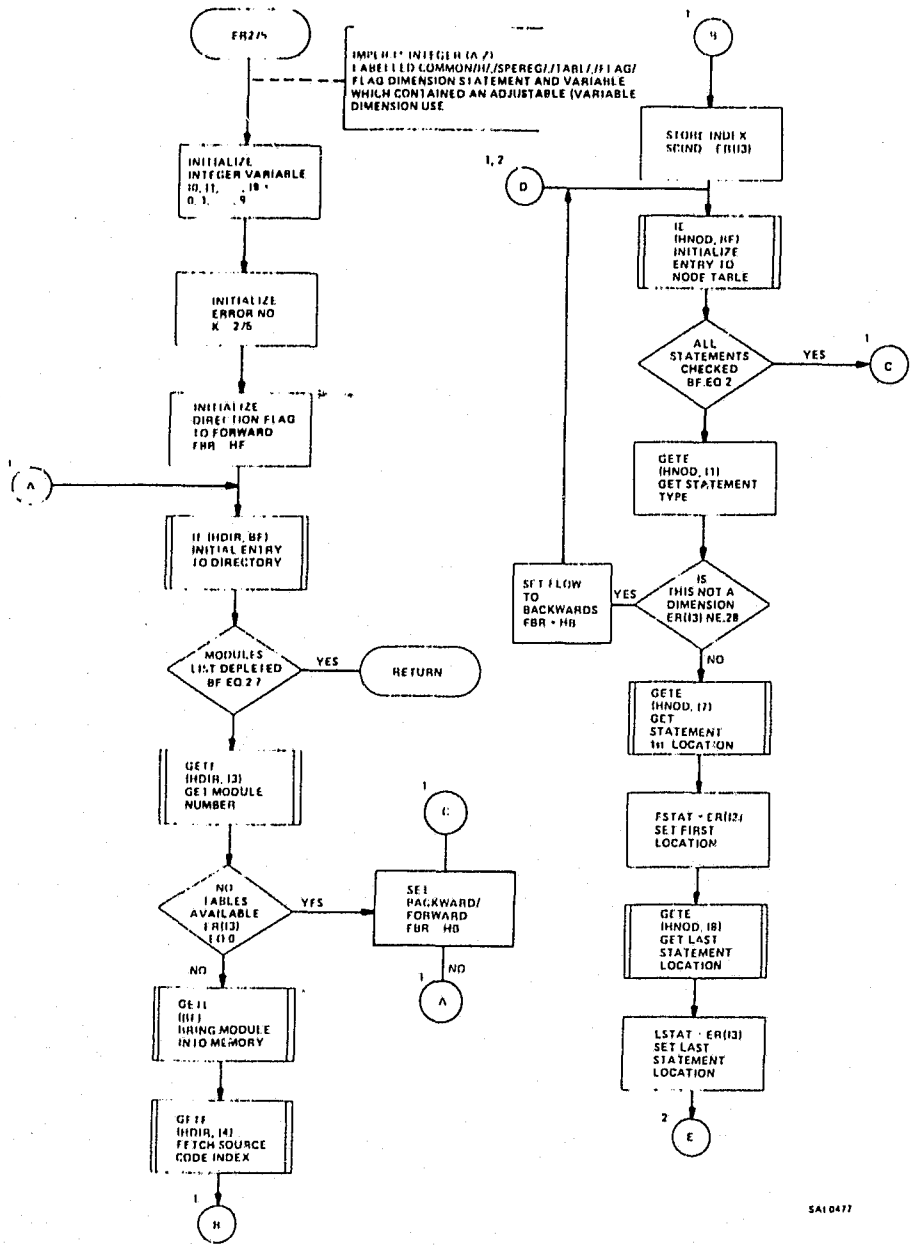
NUMOCC -- counter for error

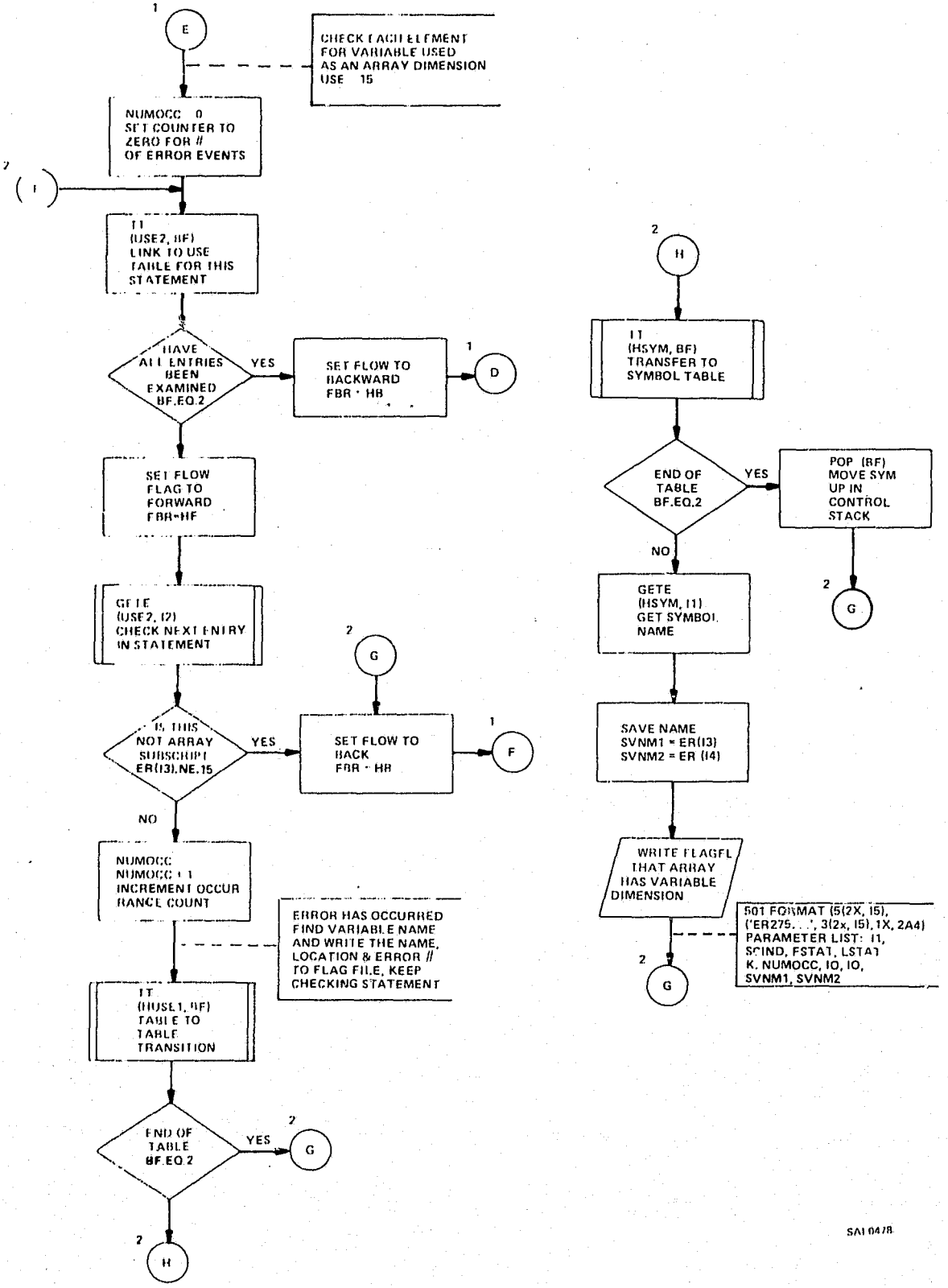
RESTRICTIONS

All variables are IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

Attached





SAI 0478

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

UNIT MODULE DESCRIPTION

IDENTIFICATION

ER280

STORAGE ALLOCATION (estimate)

PURPOSE

Flags subroutine calls with constants and hollerith arguments and flags all occurences where the same variable exists in multiple positions.

DESCRIPTION

This routine searches the parameter list of each CALL statement (statement type 34) for either equal variable symbols or constants (symbol class 8) and holleriths (symbol type 6) used as an entire argument (USE table use code 19 and 17).

HOW ENTERED

Called by AIR subroutine

CALLING SEQUENCE

Call ER280 (QNUM)
QNUM is the query number = 280 or 285

OTHER ROUTINES CALLED

IE
GETE
GETL
CONALP
POP

SET/USE PARAMETERS

USE

GLOBAL: COMMON/ALINFO/

COMMON/FLAG/

FLAGFL -- input/output designator

COMMON /H/

HB -- hollerith B
HF -- hollerith F

COMMON/SPEREG/

ER(10) -- Error registers

NOTE: Only ER(3) and ER(4) are set.

FBR -- forward/backward register

COMMON/TABLE/

HDIR -- directory table
HUSE1 -- use table
HSYM -- symbol table
HNOD -- node table

SIGNIFICANT INTERNAL VARIABLES

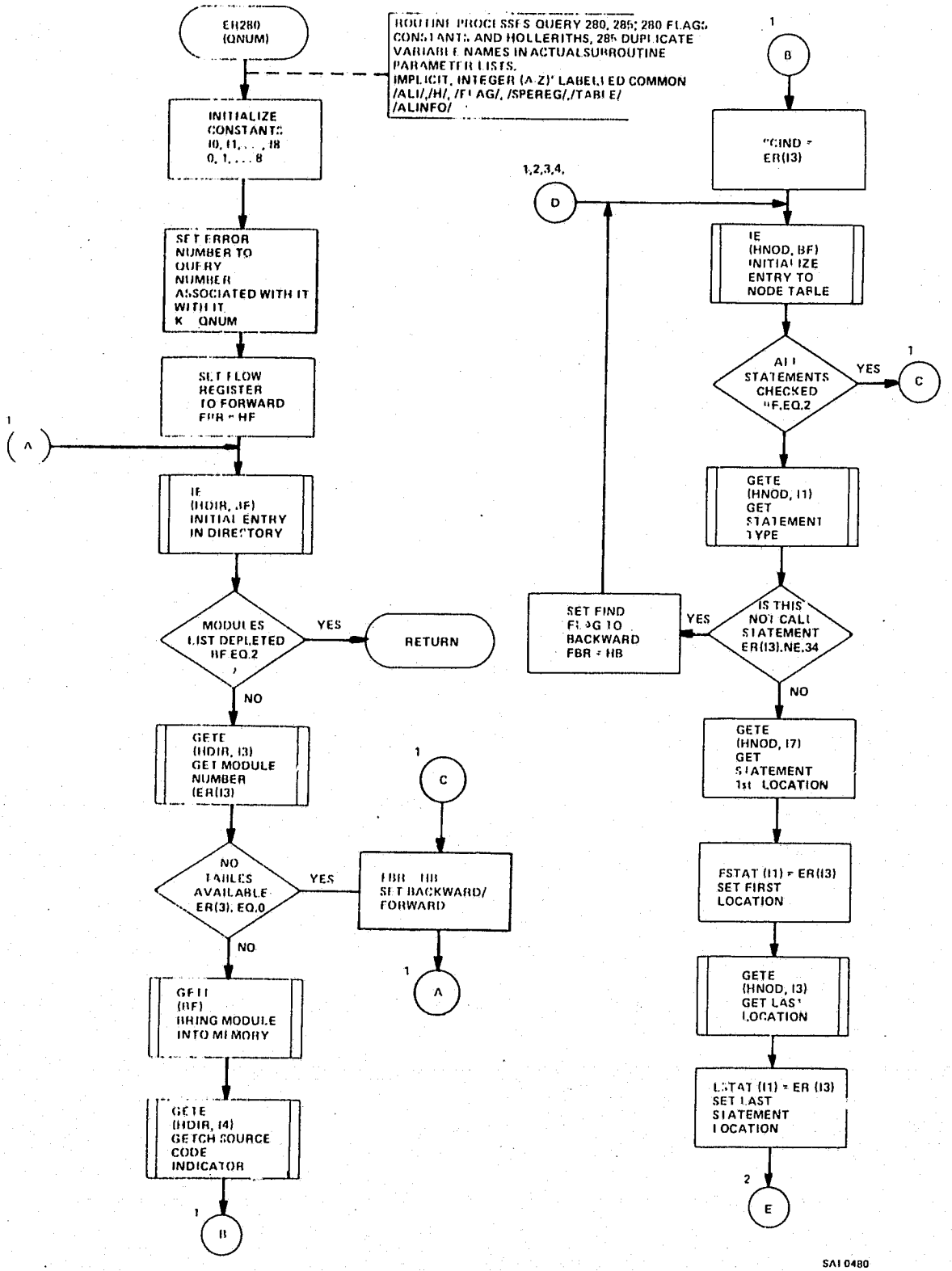
OVFLG -- Output variable from CONALP; table overflow
ERFLG -- Output variable from CONALP; error flag
PTR -- Input variable to CONALP; pointer to statement
I0, I1, ..., I9 -- represent integers 1 - 9
K -- error flag
SCIND -- source code indicator
FSTAT -- first statement number
LSTAT -- last statement number
BF -- general purpose flag

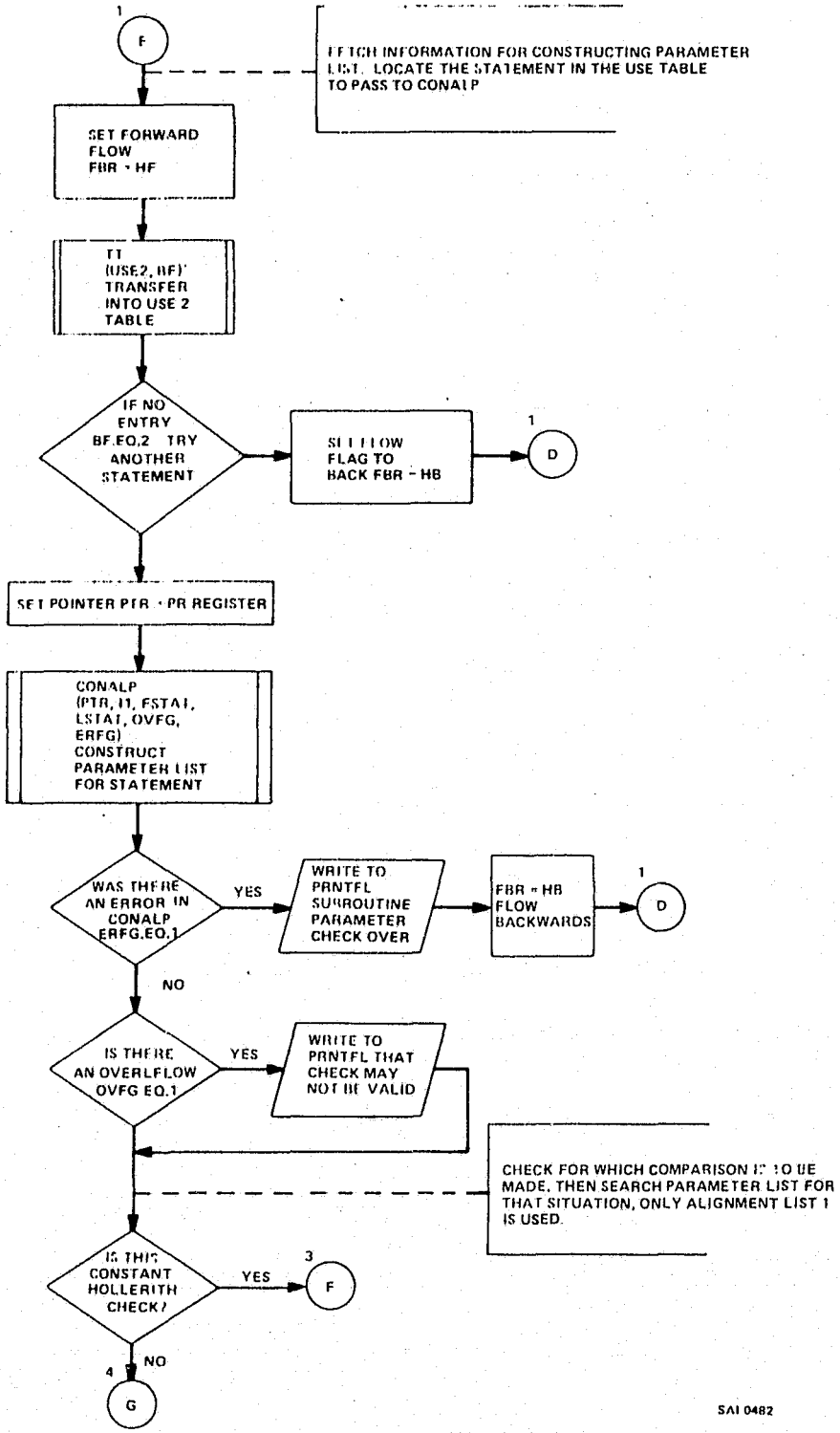
LIMITATIONS AND RESTRICTIONS

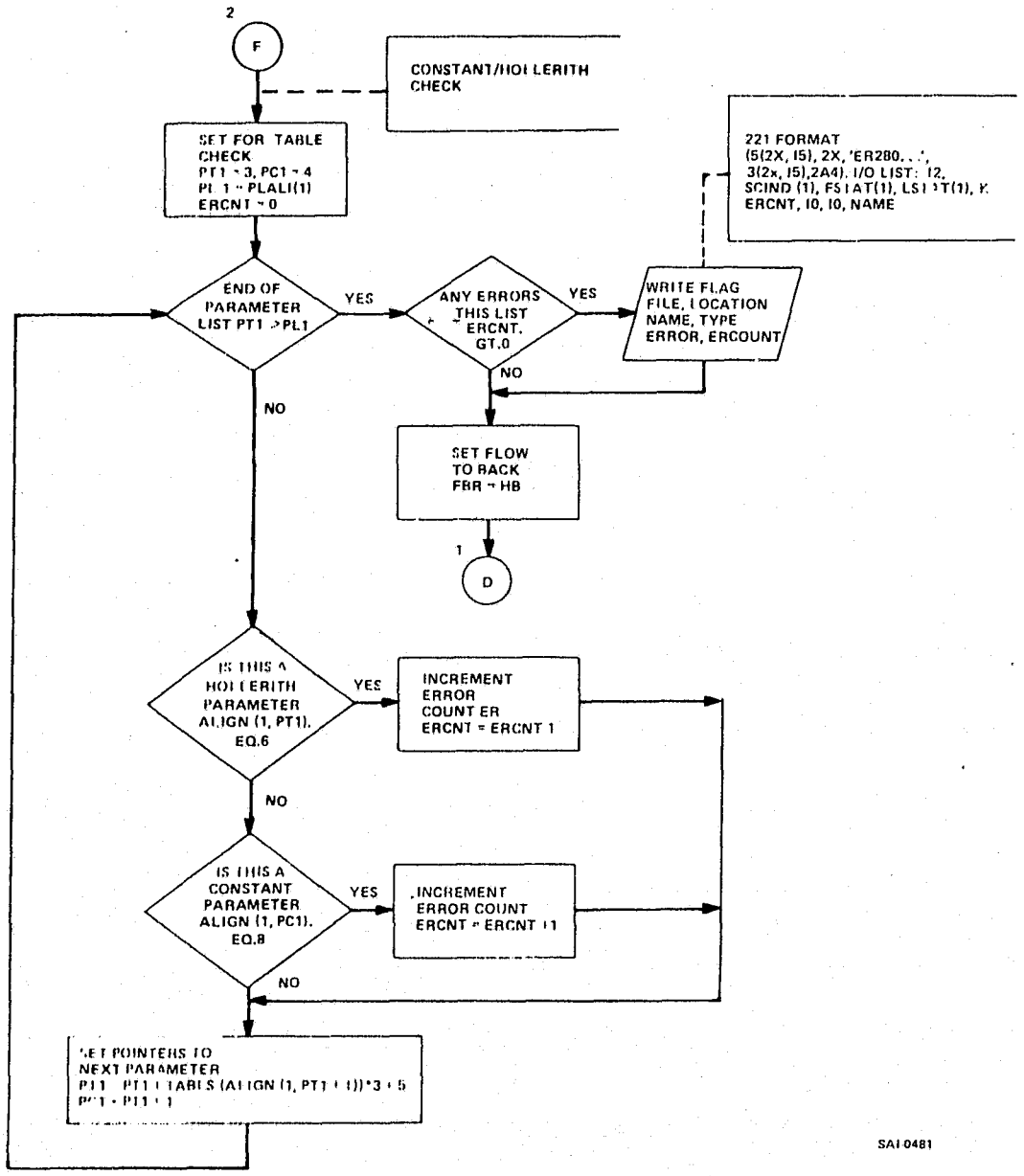
All variables are set by IMPLICIT INTEGER (A-Z)

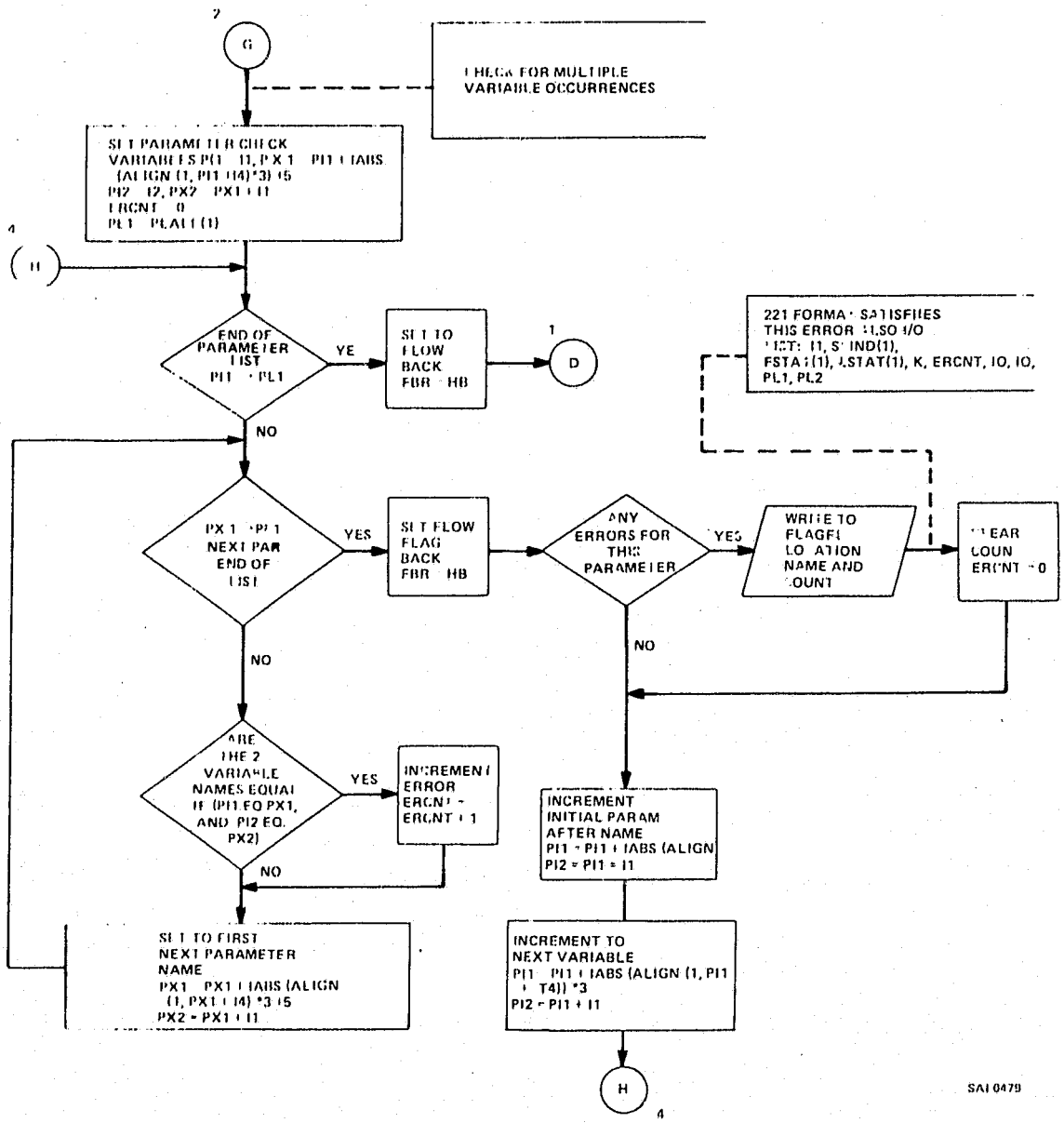
DETAILED FLOWCHART

Attached




1
F
FETCH INFORMATION FOR CONSTRUCTING PARAMETER LIST. LOCATE THE STATEMENT IN THE USE TABLE TO PASS TO CONALP
SET FORWARD FLOW FBR = HF
TT (USE2, HF) TRANSFER INTO USE 2 TABLE
IF NO ENTRY BF.EQ.2 TRY ANOTHER STATEMENT
SET FLOW FLAG TO BACK FBR = HB
1
D
SET POINTER PTR = PR REGISTER
CONALP (PTR, I1, FSTAT, LSTAT, OVFG, ERFG) CONSTRUCT PARAMETER LIST FOR STATEMENT
WAS THERE AN ERROR IN CONALP ERFG.EQ.1
YES
WRITE TO PRNTFL SUBROUTINE PARAMETER CHECK OVER
FBR = HB FLOW BACKWARDS
1
D
NO
IS THERE AN OVERLFLOW OVFG EQ.1
YES
WRITE TO PRNTFL THAT CHECK MAY NOT BE VALID
CHECK FOR WHICH COMPARISON IS TO BE MADE, THEN SEARCH PARAMETER LIST FOR THAT SITUATION, ONLY ALIGNMENT LIST 1 IS USED.
IS THIS CONSTANT HOLLERITH CHECK?
YES
3
F
4
NO
G
SAI 0482


2
F
CONSTANT/HOLLERITH CHECK
SET FOR TABLE CHECK
PT1 ← 3, PC1 ← 4
PL1 ← PLALI(1)
ERCNT ← 0
221 FORMAT
(5(2X, I5), 2X, 'ER280. . .',
3(2x, I5),2A4), I/O LIST: I2,
SCIND (1), FSTAT(1), LSTAT(1), P.
ERCNT, I0, I0, NAME
END OF PARAMETER LIST PT1 > PL1
YES
ANY ERRORS THIS LIST ERCNT. GT.0
YES
WRITE FLAG FILE, LOCATION NAME, TYPE ERROR, ERCOUNT
NO
NO
SET FLOW TO BACK FBR ← HB
1
D
IS THIS A HOLLERITH PARAMETER ALIGN (1, PT1). EQ.6
YES
INCREMENT ERROR COUNT ER ERCNT = ERCNT 1
NO
IS THIS A CONSTANT PARAMETER ALIGN (1, PC1). EQ.8
YES
INCREMENT ERROR COUNT ERCNT = ERCNT 1
NO
SET POINTERS TO NEXT PARAMETER
PT1 = PT1 + TABLES (ALIGN (1, PT1 + 1))*3 + 5
PC1 = PT1 + 1
SAI 0481



UNIT MODULE DESCRIPTION

IDENTIFICATION

MULBRA additions

STORAGE ALLOCATION REQUIREMENT (estimate)

Total after additions 2K hexadecimal bytes

PURPOSE

Flag arithmetic IFs and flag GO TOs that are targets of other GO TOs.

DESCRIPTION

One addition to MULBRA will flag GO TO statements which are the targets of previous GO TO statements. The program will search for statement type 45 that has a use code of 9. When this occurs the statement is flagged with a message to the Flag File (220).

The second addition to MULBRA flags Arithmetic IFs. In this case an IF statement (statement type 10) with more than 2 branch targets is flagged by an error message to the flag file (220).

HOW ENTERED

Called by AIR upon query request.

CALLING SEQUENCE

CALL MULBRA (Number)
Number is query number 150, 200, or 220; all are processed in this subroutine.

OTHER ROUTINES CALLED

No additional routines are called.

SET/USE PARAMETERS

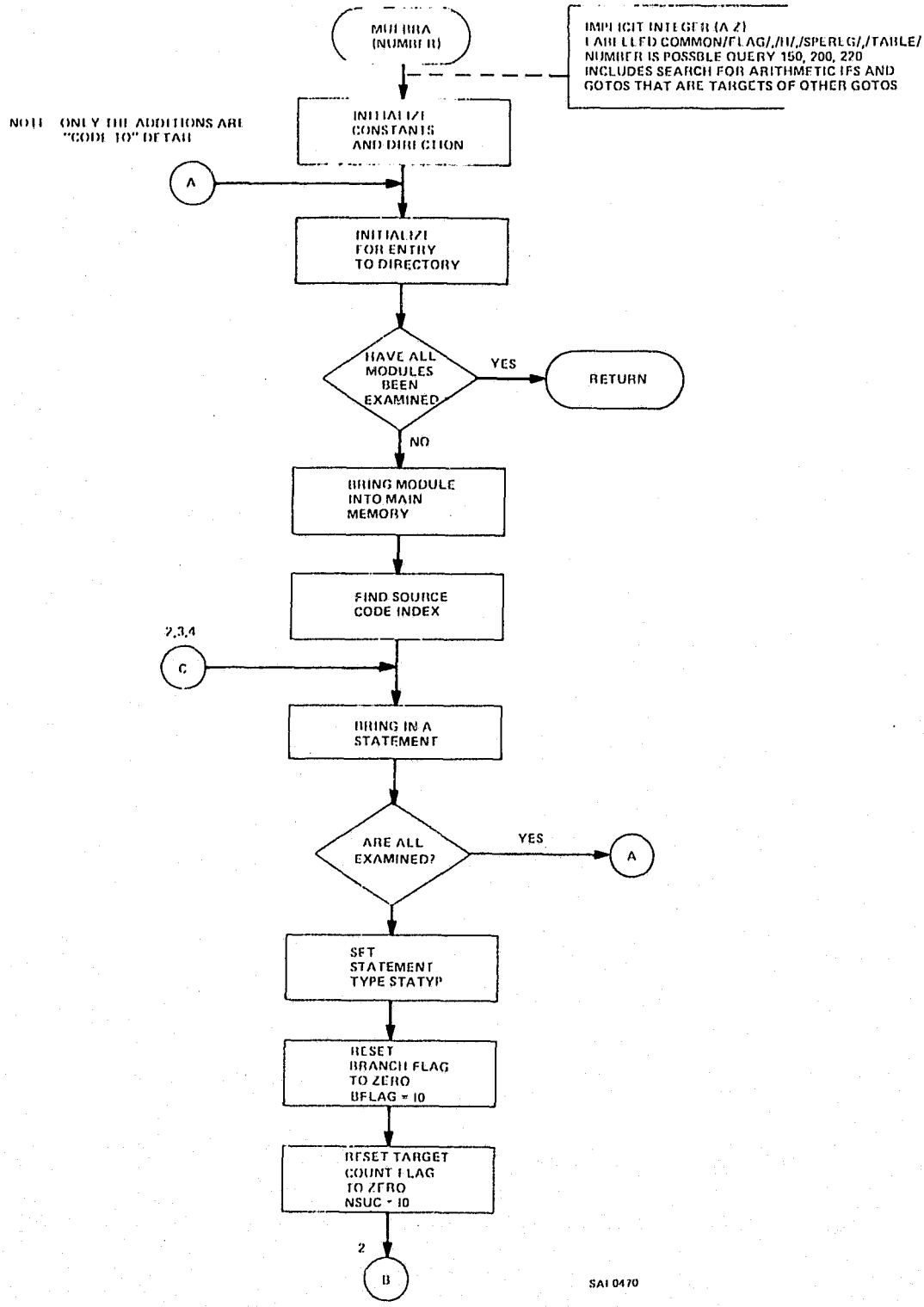
There are no additional parameters used or set.

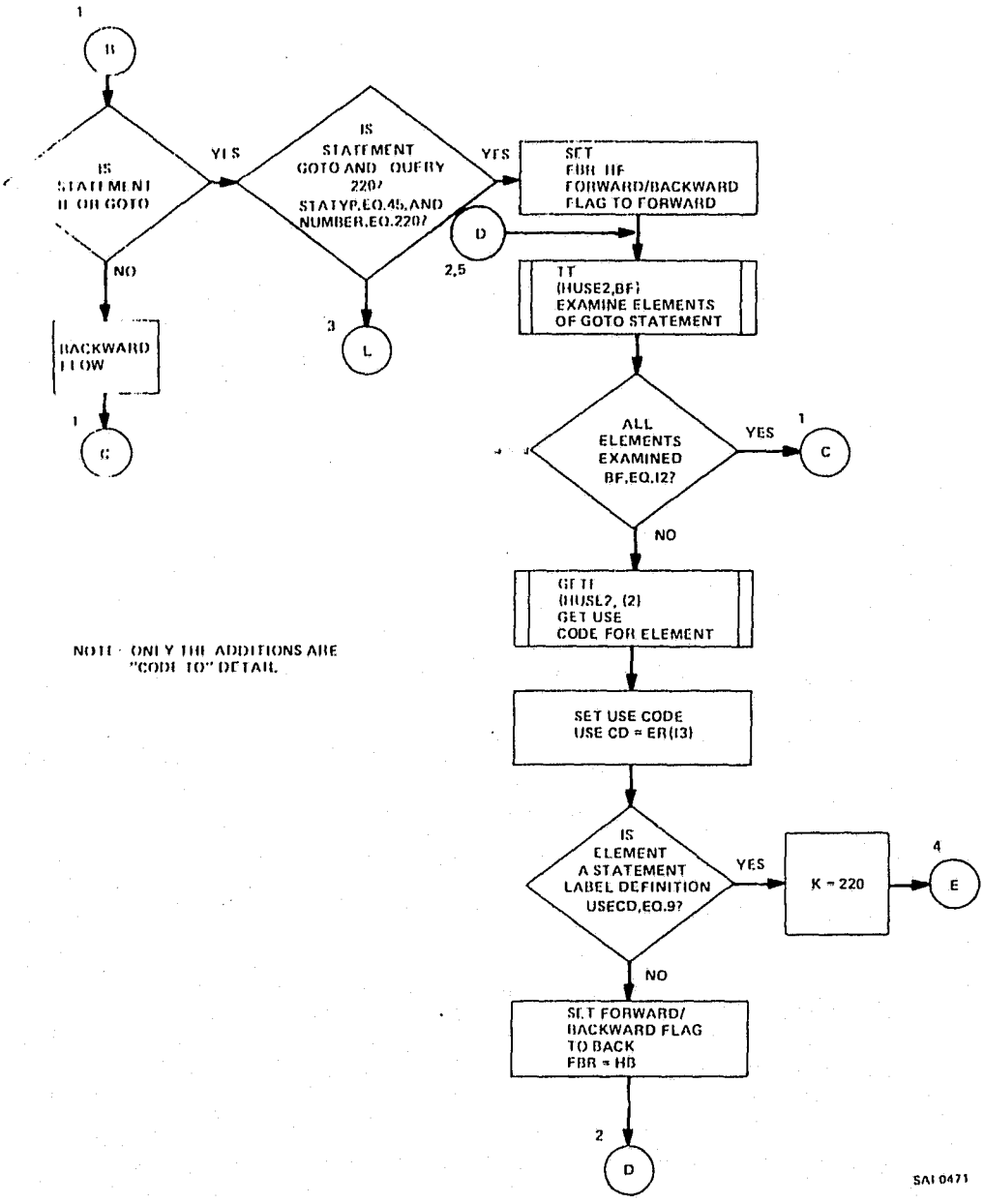
SIGNIFICANT INTERNAL VARIABLES

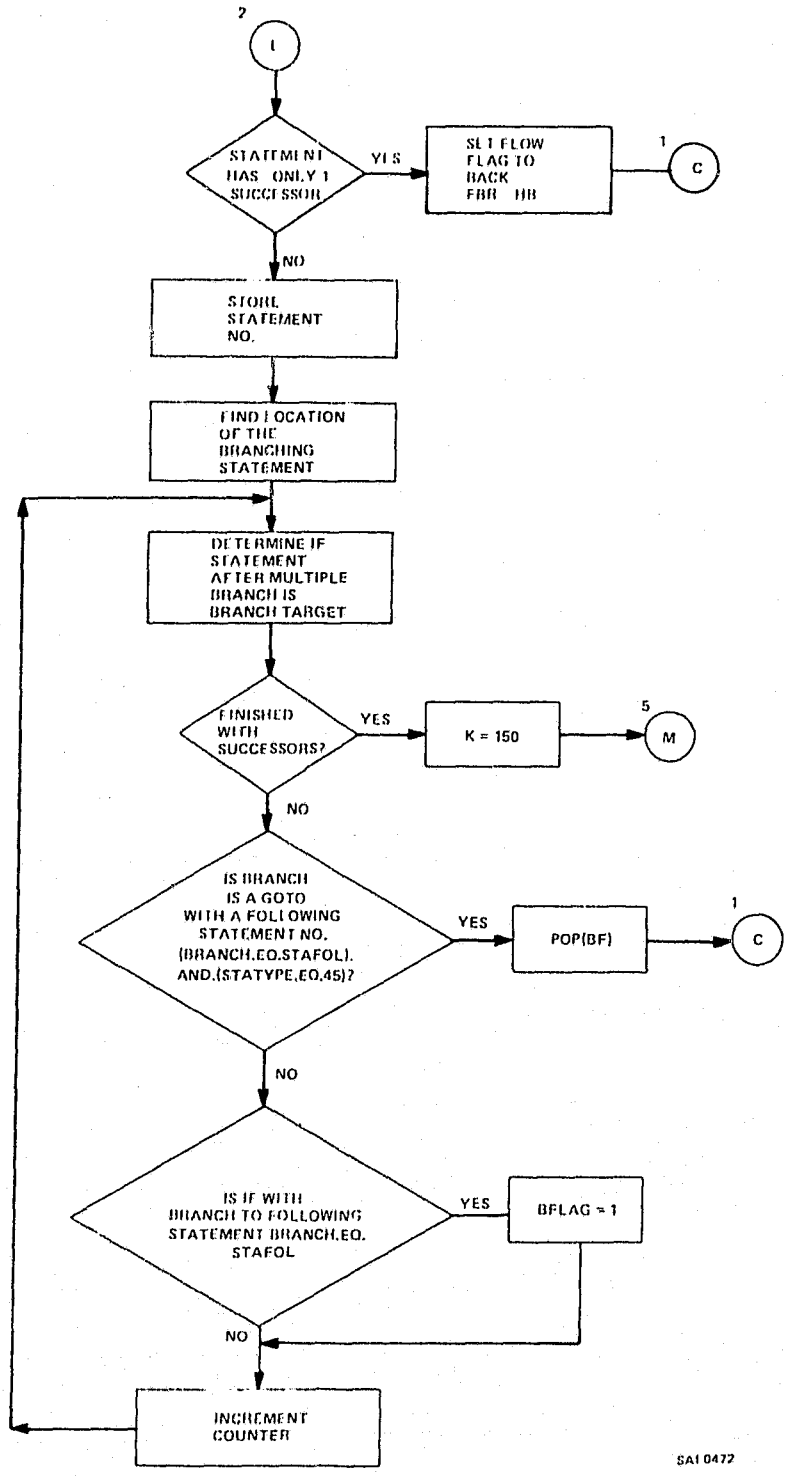
USECD -- use code variable
NSUC -- number of targets
BFLAG -- branch flag

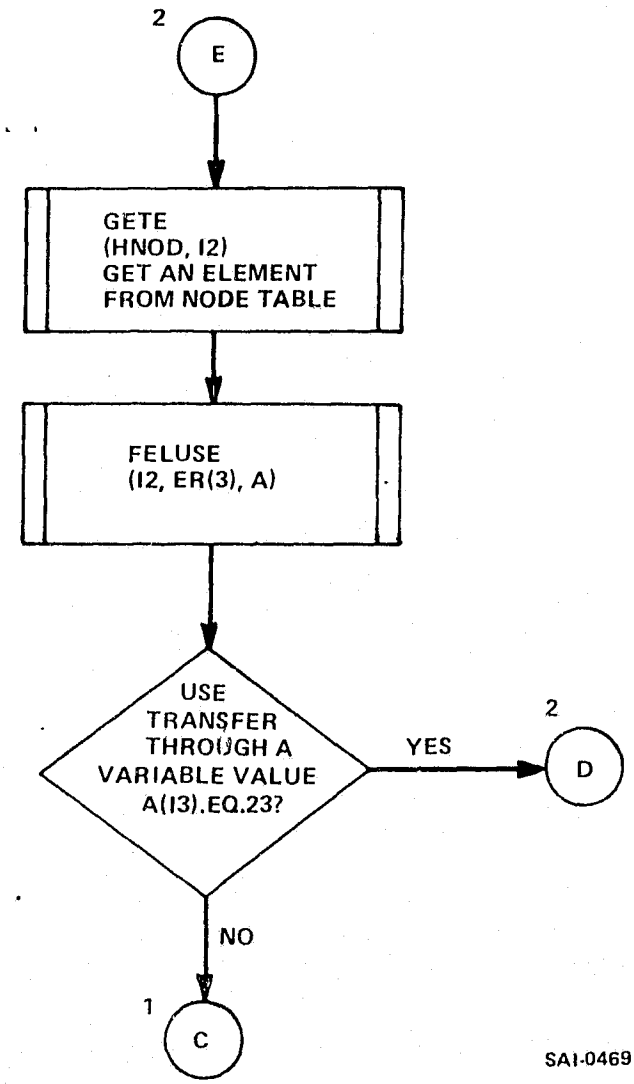
DETAILED FLOWCHART

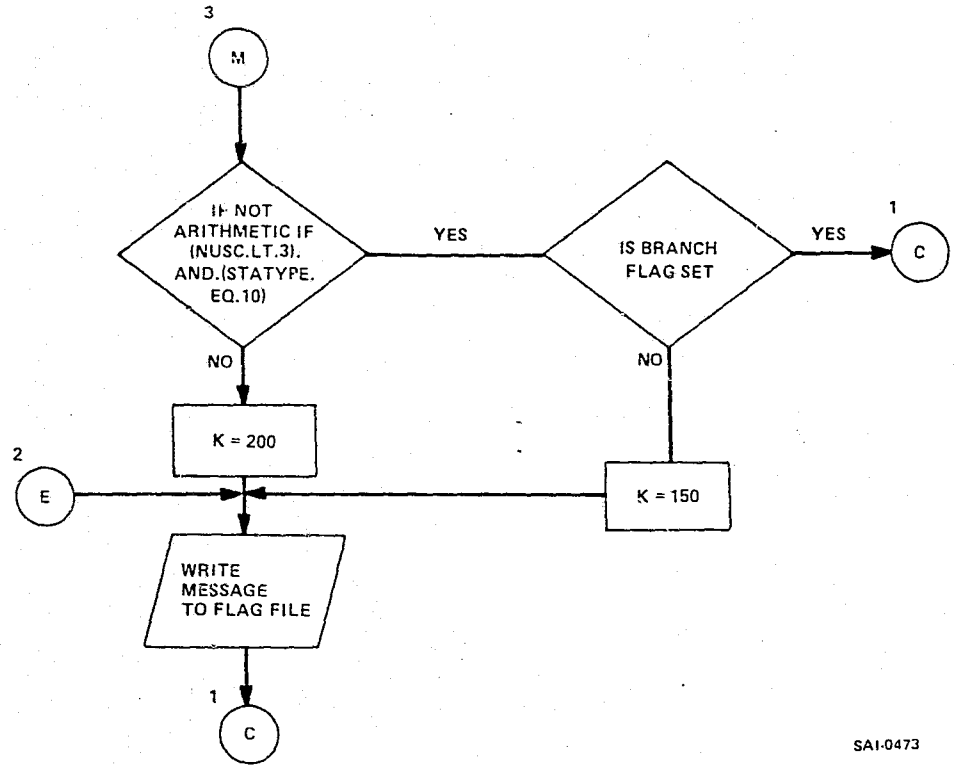
Attached




1
B
IS STATEMENT II OR GOTO
YES
IS STATEMENT GOTO AND QUERY 220? STATYP.EQ.45,AND NUMBER.EQ.220?
YES
SET FBR TO FORWARD/BACKWARD FLAG TO FORWARD
D
2,5
NO
BACKWARD FLOW
3
L
1
C
TT (HUSE2,BF) EXAMINE ELEMENTS OF GOTO STATEMENT
ALL ELEMENTS EXAMINED BF.EQ.12?
YES
1
C
NO
GETT (HUSL2, (2) GET USE CODE FOR ELEMENT
SET USE CODE USE CD = ER(13)
IS ELEMENT A STATEMENT LABEL DEFINITION USECD,EQ.9?
YES
K = 220
4
E
NO
SET FORWARD/ BACKWARD FLAG TO BACK FBR = HB
2
D
NOTE - ONLY THE ADDITIONS ARE "CODE TO" DETAIL
SAI 0471




2
E
GETE
(HNOD, I2)
GET AN ELEMENT
FROM NODE TABLE
FELUSE
(I2, ER(3), A)
USE
TRANSFER
THROUGH A
VARIABLE VALUE
A(I3).EQ.23?
YES
2
D
NO
1
C
SAI-0469


3
M
IF NOT ARITHMETIC IF (NUSC.LT.3). AND.(STATYPE. EQ.10)
YES
IS BRANCH FLAG SET
YES
1
C
NO
NO
K = 200
2
E
K = 150
WRITE MESSAGE TO FLAG FILE
1
C
SAI-0473

UNIT MODULE DESCRIPTION

IDENTIFICATION

ER250 -- Error 250 Routine

STORAGE ALLOCATION (estimate)

2K (hexadecimal bytes)

PURPOSE

To flag variables used as I/O designators.

DESCRIPTION

This routine examines each line of code in the source decks for variables with the classification "scalar" (class code = 6) which are also used as I/O designators (use code = 26). Having found one which satisfies both requirements, an applicable message is written to the flag file.

MATHEMATICAL EQUATIONS/DEFINITIONS

None

HOW ENTERED

Called by AIR

CALLING SEQUENCE

Subroutine call, no arguments. Call ER250.

UNIT MODULE OR OTHER ROUTINES CALLED

IE
GETE
GETL
TT

SET/USE PARAMETERS

SET

Global Common: COMMON/SPEREG/

ER(10) -- Error registers
NOTE: Only ER(3) and ER(4) are set.

FBR -- forward/backward register

USE

Global Common: COMMON/FLAG/

FLAGFL -- input/output designator

Global Common: COMMON/H/

HB -- hollerith B
HF -- hollerith F

Global Common: COMMON/TABLE/

HDIR -- directory table
HUSE1 -- use table
HSYM -- symbol table
HNOD -- node table

SIGNIFICANT INTERNAL VARIABLES

I0, I1, . . . , I9 -- represent integers 1 - 9
K -- error flag
SCIND -- source code indicator
FSTAT -- first statement number
LSTAT -- last statement number
BF -- general purpose flag

CONSTRAINTS

All variables are set to integer by the IMPLICIT statement.

COMPUTATIONAL ACCURACY AND RANGE OF INPUT/OUTPUT VARIABLES

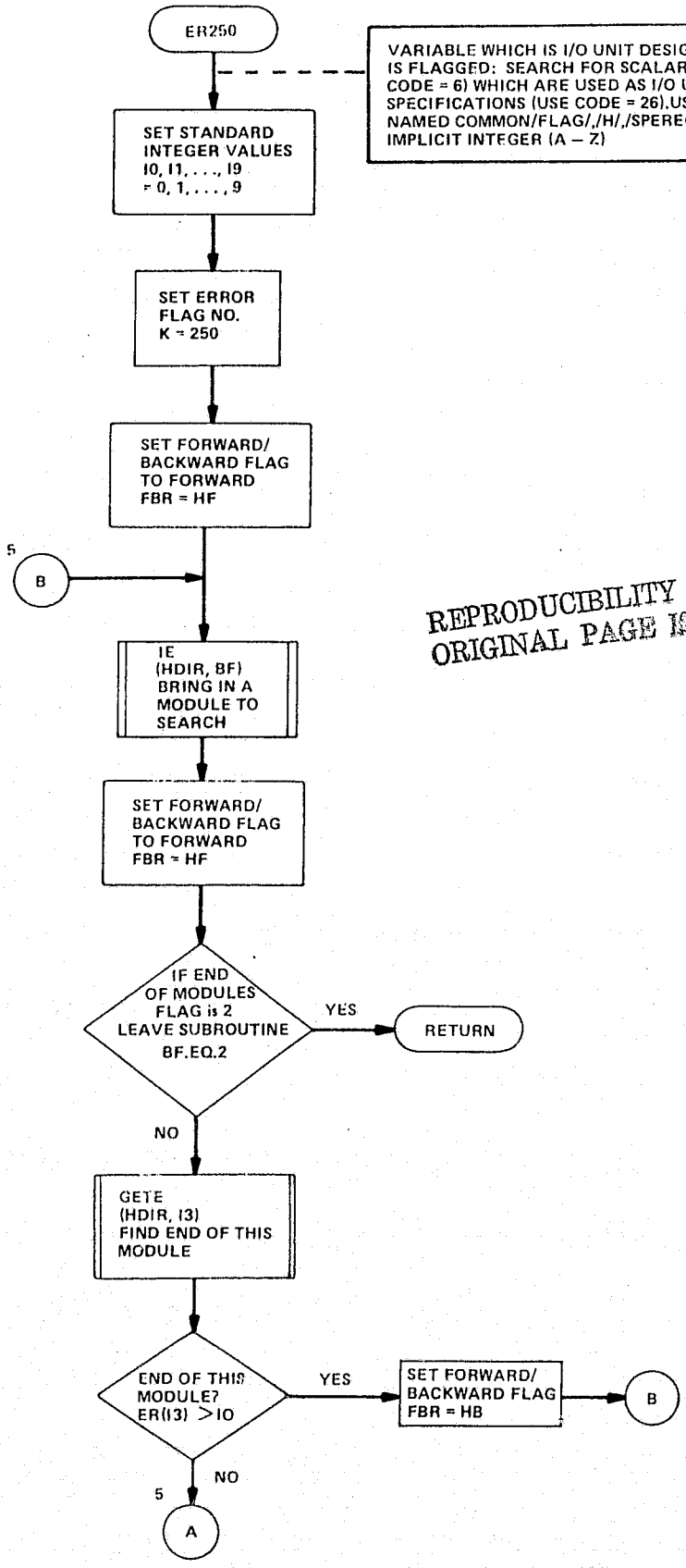
Non-Applicable

ERROR MESSAGES AND SUMMARY

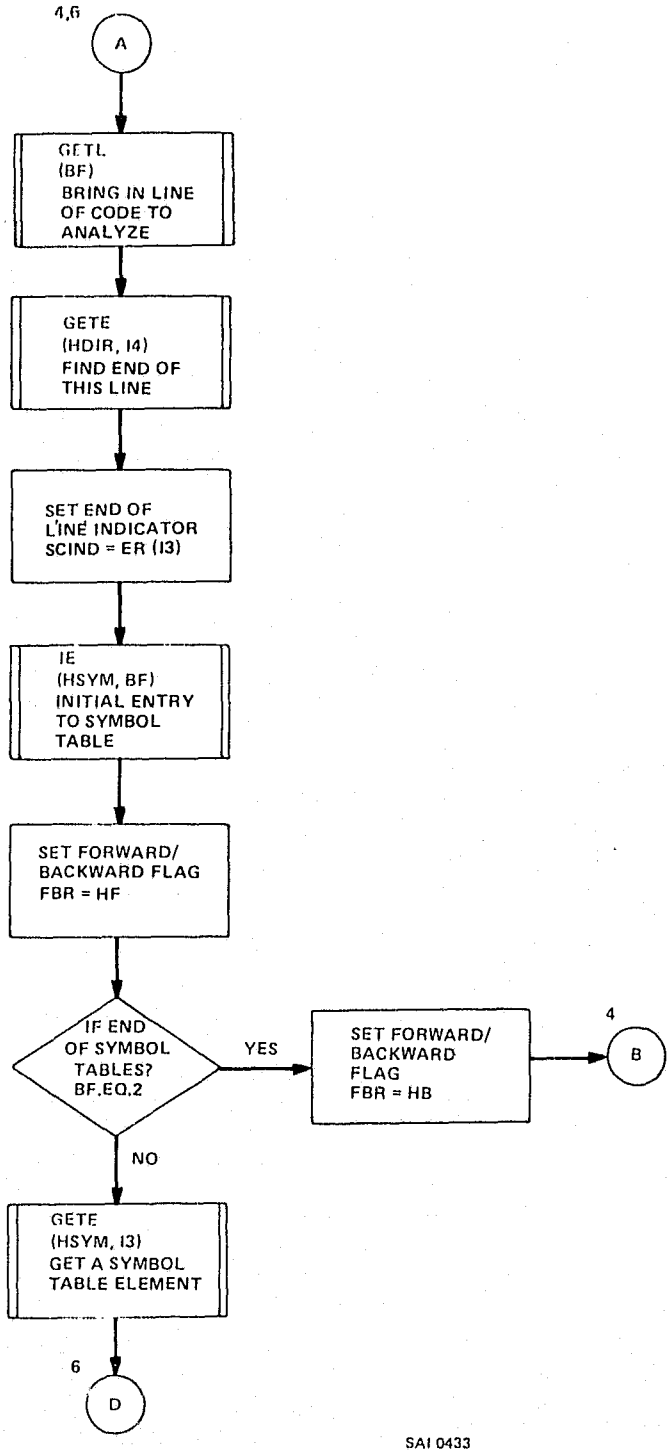
None

DETAILED FLOWCHART

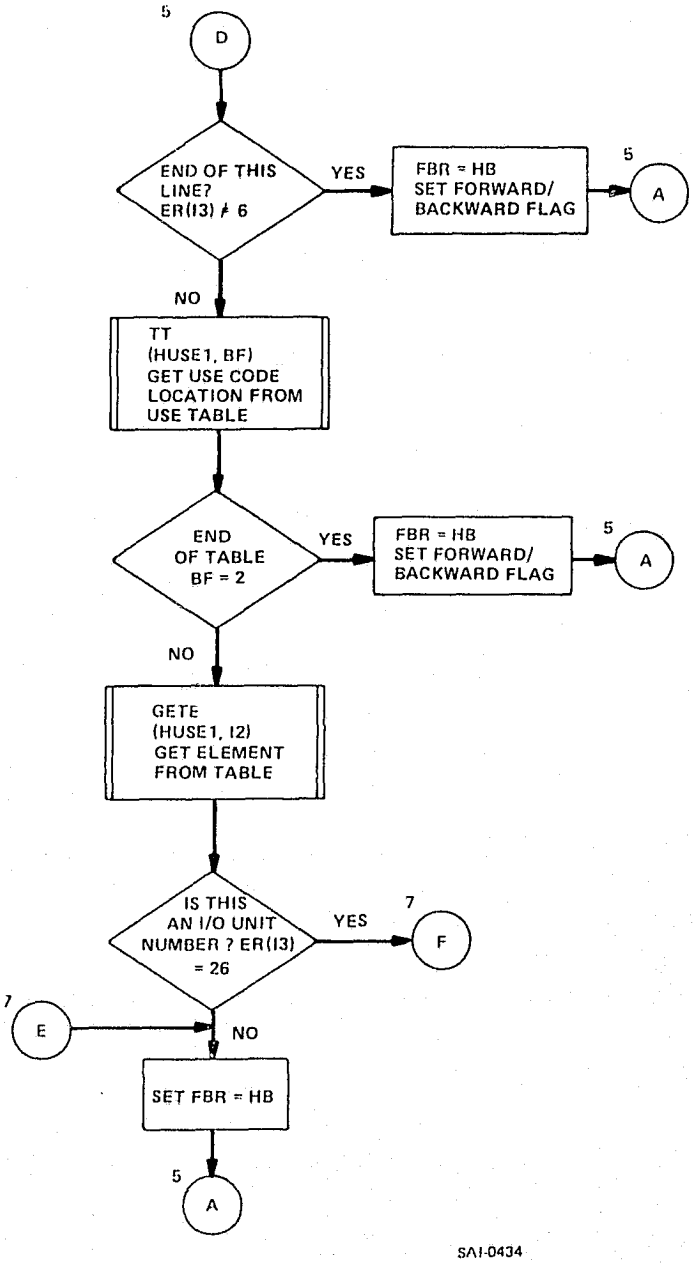
See following pages.

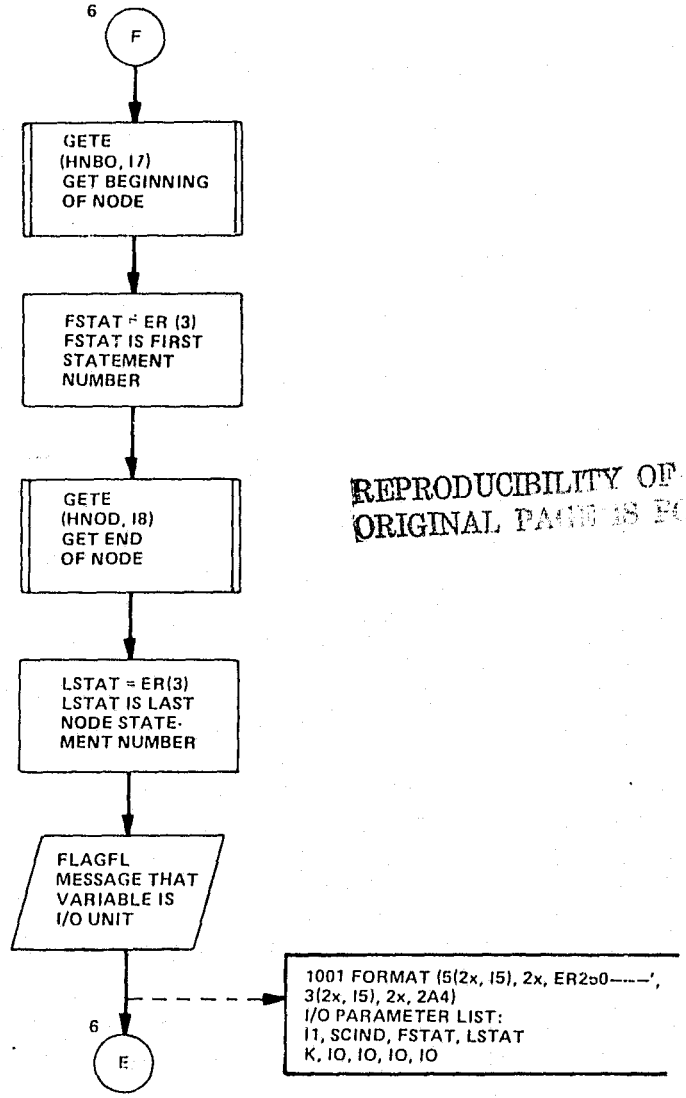


SAI-0432


4,6
A
GETL
(BF)
BRING IN LINE
OF CODE TO
ANALYZE
GETE
(HDIR, I4)
FIND END OF
THIS LINE
SET END OF
LINE INDICATOR
SCIND = ER (I3)
IE
(HSYM, BF)
INITIAL ENTRY
TO SYMBOL
TABLE
SET FORWARD/
BACKWARD FLAG
FBR = HF
IF END
OF SYMBOL
TABLES?
BF.EQ.2
YES
SET FORWARD/
BACKWARD
FLAG
FBR = HB
4
B
NO
GETE
(HSYM, I3)
GET A SYMBOL
TABLE ELEMENT
6
D

SAI 0433


5
D
END OF THIS LINE? ER(I3) ≠ 6
YES
FBR = HB
SET FORWARD/ BACKWARD FLAG
5
A
NO
TT
(HUSE1, BF)
GET USE CODE LOCATION FROM USE TABLE
END OF TABLE BF = 2
YES
FBR = HB
SET FORWARD/ BACKWARD FLAG
5
A
NO
GETE
(HUSE1, I2)
GET ELEMENT FROM TABLE
IS THIS AN I/O UNIT NUMBER ? ER(I3) = 26
YES
7
F
7
E
NO
SET FBR = HB
5
A
SAI-0434



SAI-0435

UNIT MODULE DESCRIPTION

IDENTIFICATION

ER260 -- Error 260 routine

STORAGE ALLOCATION (estimate)

2K (hexadecimal bytes)

PURPOSE

Flag statement labels not in increasing numerical order.

DESCRIPTION

This routine locates and reconstructs each statement label in a module. If the current label is less than the previous label it is flagged.

HOW ENTERED

Called by AIR

CALLING SEQUENCE

Subroutine: Call ER260
There are no calling arguments

UNIT MODULE OR OTHER ROUTINES CALLED

SUBROUTINE:	IE
	GETE
	GETL
	CONVER
FUNCTION:	FLD

SET/USE PARAMETERS

SET	USE
Global: COMMON/SPEREG/	GLOBAL: COMMON/FLAG/
ER(10)	Flagfl -- I/O file
FBR	
NOTE: Only ER(3) is set, the rest are not referenced.	COMMON/HOSTWD/
	FULLWD
	COMMON/H/
	HB -- hollerith B
	HF -- hollerith F
	COMMON/TABLE/
	HDIR
	HSYM
	HNOD

SIGNIFICANT INTERNAL VARIABLES

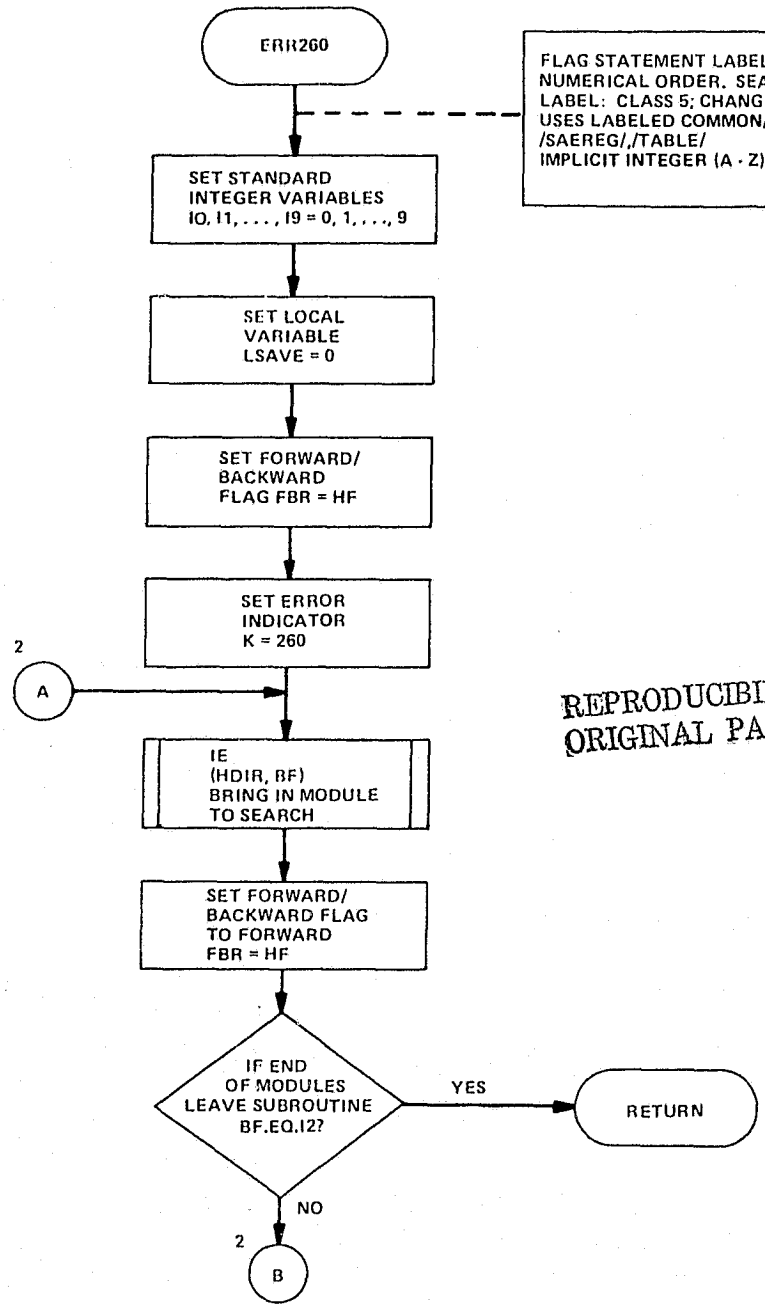
```
I0, I1, ... I9 -- integers 1 - 9
LSAVE          -- current label number
K              -- error number flag
SCIND          -- source code indicator
FSTAT          -- first statement number of error
LSTAT          -- last statement number of error
BF             -- condition flag; return argument
LABEL          -- next reconstructed label number
```

LIMITATIONS AND RESTRICTIONS

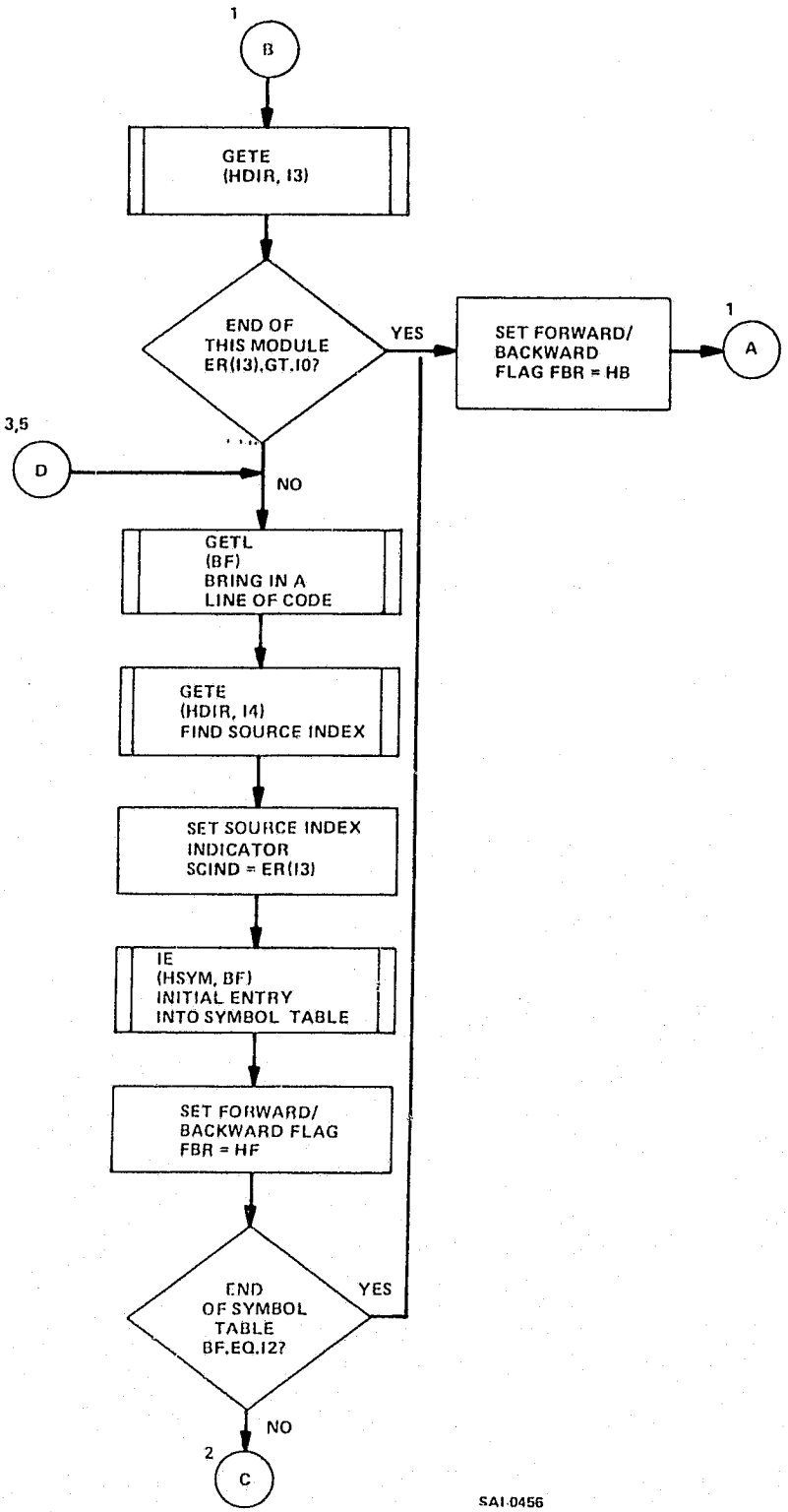
All variables are set to integer by the IMPLICIT statement.

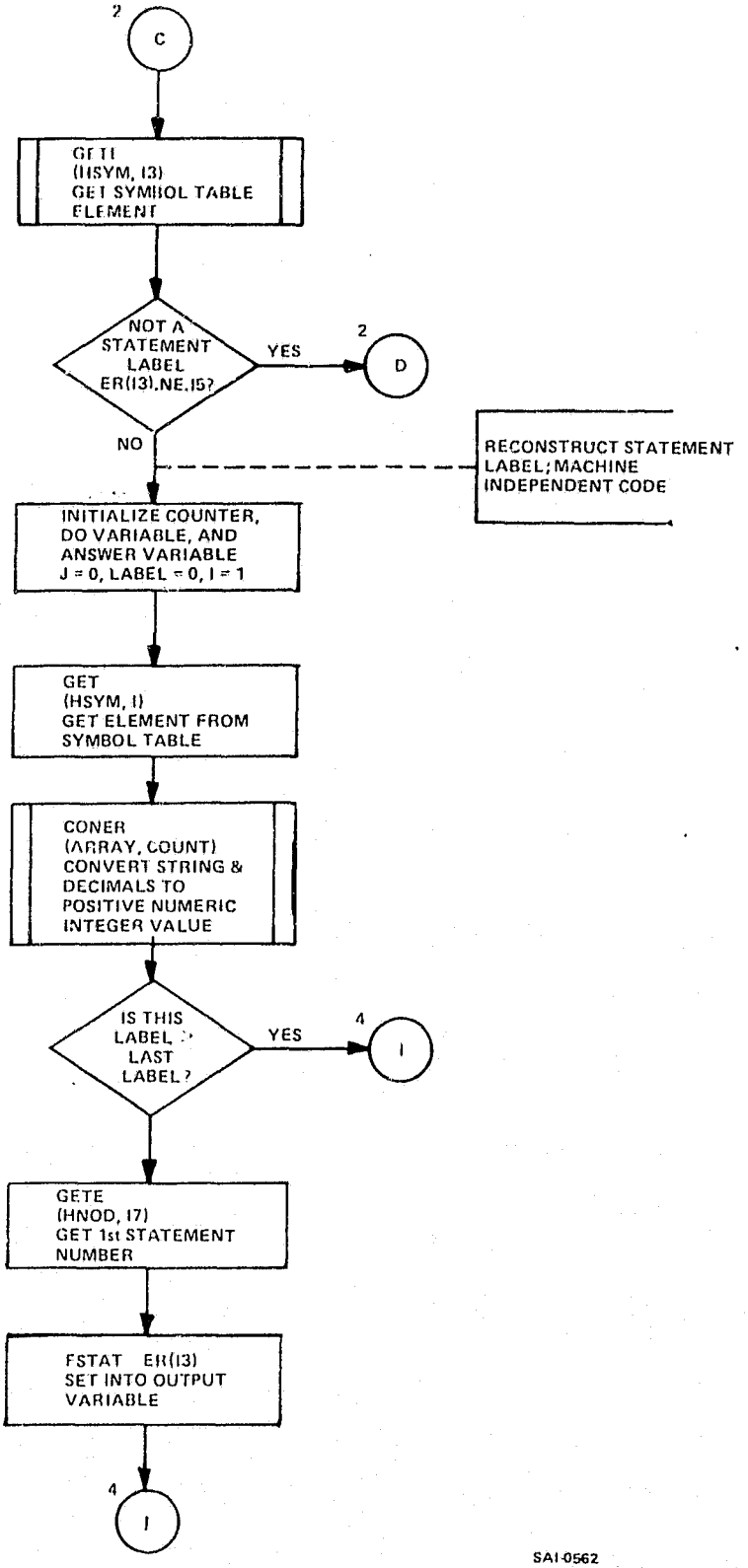
DETAILED FLOWCHART

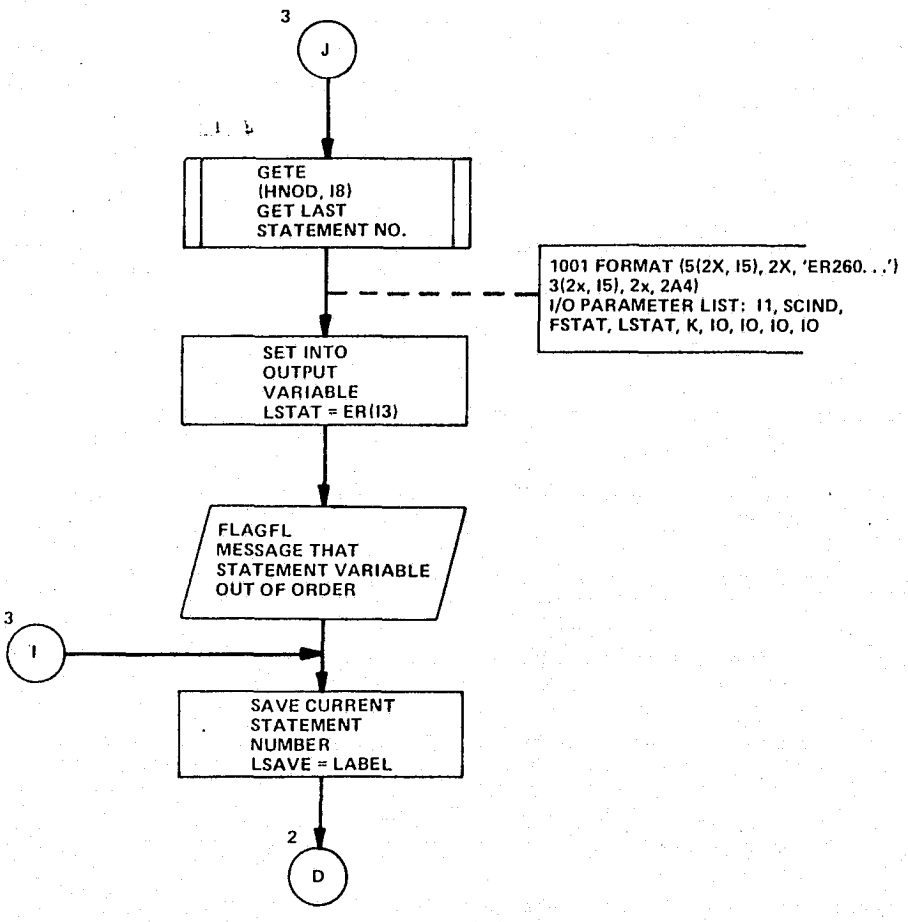
See Attached



SAI-0458


1
B
GETE
(HDIR, I3)
END OF
THIS MODULE
ER(I3).GT.I0?
YES
SET FORWARD/
BACKWARD
FLAG FBR = HB
1
A
3,5
D
NO
GETL
(BF)
BRING IN A
LINE OF CODE
GETE
(HDIR, I4)
FIND SOURCE INDEX
SET SOURCE INDEX
INDICATOR
SCIND = ER(I3)
IE
(HSYM, BF)
INITIAL ENTRY
INTO SYMBOL TABLE
SET FORWARD/
BACKWARD FLAG
FBR = HF
END
OF SYMBOL
TABLE
BF.EQ.I2?
YES
NO
2
C
SAI-0456


2
C
GET
(HSYM, I3)
GET SYMBOL TABLE
ELEMENT
NOT A
STATEMENT
LABEL
ER(I3).NE.I5?
YES
2
D
NO
RECONSTRUCT STATEMENT
LABEL; MACHINE
INDEPENDENT CODE
INITIALIZE COUNTER,
DO VARIABLE, AND
ANSWER VARIABLE
J = 0, LABEL = 0, I = 1
GET
(HSYM, I)
GET ELEMENT FROM
SYMBOL TABLE
CONER
(ARRAY, COUNT)
CONVERT STRING &
DECIMALS TO
POSITIVE NUMERIC
INTEGER VALUE
IS THIS
LABEL
LAST
LABEL?
YES
4
I
GETE
(HNOD, I7)
GET 1st STATEMENT
NUMBER
FSTAT ER(I3)
SET INTO OUTPUT
VARIABLE
4
I
SAI-0562


3
J
GETE
(HNOD, I8)
GET LAST
STATEMENT NO.
1001 FORMAT (5(2X, I5), 2X, 'ER260. . .')
3(2x, I5), 2x, 2A4)
I/O PARAMETER LIST: I1, SCIND,
FSTAT, LSTAT, K, IO, IO, IO, IO
SET INTO
OUTPUT
VARIABLE
LSTAT = ER(I3)
FLAGFL
MESSAGE THAT
STATEMENT VARIABLE
OUT OF ORDER
3
I
SAVE CURRENT
STATEMENT
NUMBER
LSAVE = LABEL
2
D

SAI-0459

UNIT MODULE DESCRIPTION

IDENTIFICATION

ER230 Error 230 routine

STORAGE ALLOCATION (estimate)

3K (hexadecimal bytes)

PURPOSE

Flag ASSIGN, PAUSE and assigned GO TO statements

DESCRIPTION

This routine searches a module for statement 38 (ASSIGN), 50 (PAUSE), or 45 (GO TO). When the first two are found, the error is written to Flag File immediately. The third one is checked further for a use code of 23 (transfer through a variable value). If the code is 23 a message is written to the Flag File also.

HOW ENTERED

Called by AIR

CALLING SEQUENCE

CALL ER230

Note: No arguments are used.

UNIT MODULE OR OTHER ROUTINES CALLED

IE
GETE
GETL
FELUSE

SET/USE PARAMETERS

SET		USE	
GLOBAL:	COMMON/SPEREG/ ER(10) -- error registers NOTE: Only ER(3) is used FBR-- forward/ backward register	GLOBAL:	COMMON/FLAG/ FLAGFL -- I/O designator COMMON/H/ HB -- hollerith B HF -- hollerith F COMMON/TABLE/ HDIR -- directory table HNOD -- node table HSYM -- symbol table HUSE1 -- use table

SIGNIFICANT INTERNAL VARIABLES

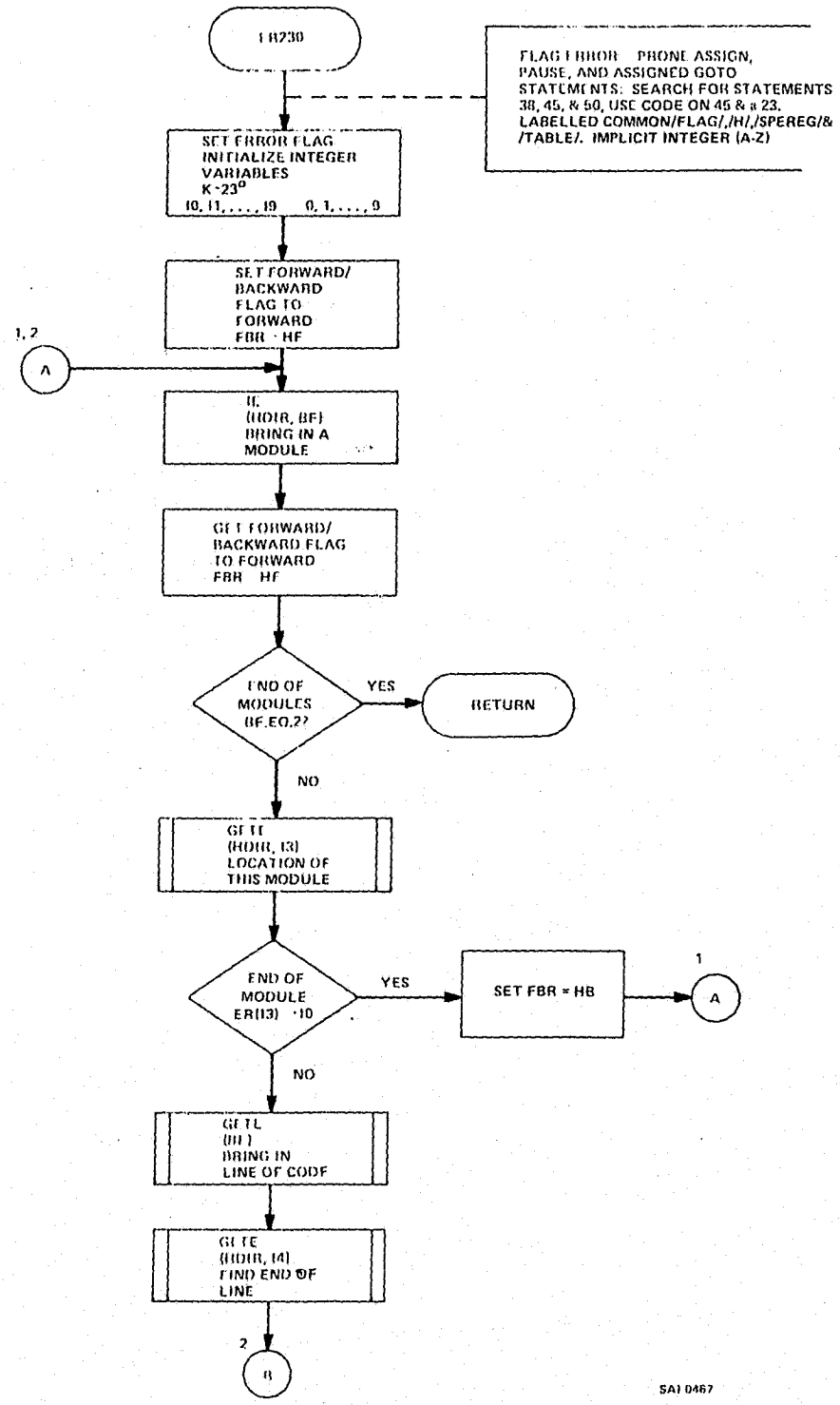
I0, I1, ..., I9	-- represent integers 1 - 9
K	-- error flag
SCIND	-- source code indicator
FSTAT	-- first statement number
LSTAT	-- last statement number
BF	-- general purpose flag
STATYP	-- statement type
A(10)	-- array for return arguments

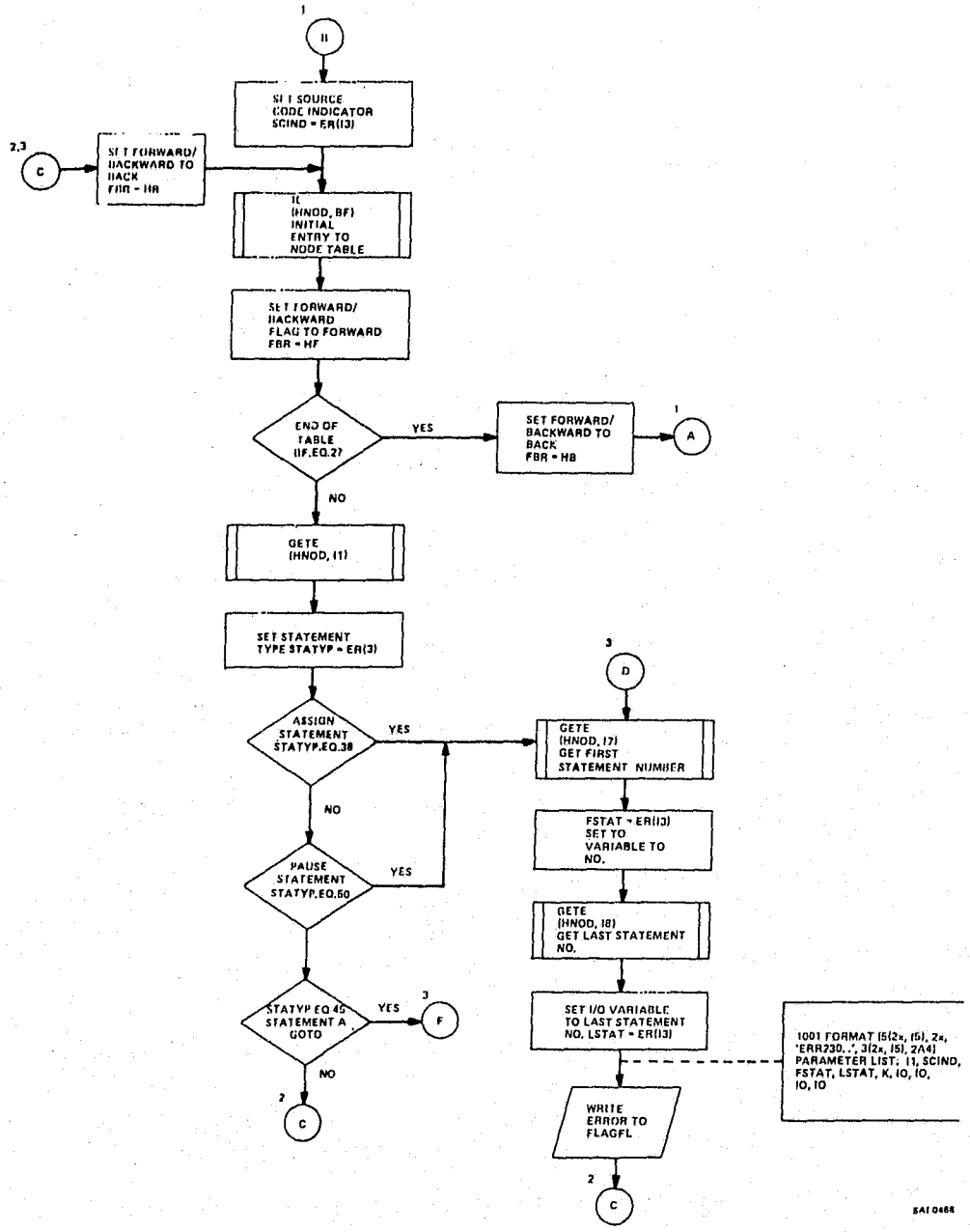
CONSTRAINTS

All variables are set to integer by IMPLICIT statement

DETAILED FLOWCHART

See attached


LIB230
FLAG ERROR PHONE ASSIGN, PAUSE, AND ASSIGNED GOTO STATEMENTS. SEARCH FOR STATEMENTS 38, 45, & 50, USE CODE ON 45 & # 23. LABELLED COMMON/FLAG/,/H/,/SPEREG/& /TABLE/. IMPLICIT INTEGER (A-Z)
SET ERROR FLAG INITIALIZE INTEGER VARIABLES K=23^0 10, 11, . . . , 19 0, 1, . . . , 9
SET FORWARD/ BACKWARD FLAG TO FORWARD FBR = HF
1, 2
A
IF (IDIR, BF) BRING IN A MODULE
GET FORWARD/ BACKWARD FLAG TO FORWARD FBR = HF
END OF MODULES BF.EQ.2?
YES
RETURN
NO
GETE (IDIR, I3) LOCATION OF THIS MODULE
END OF MODULE ER(I3) = 10
YES
SET FBR = HB
1
A
NO
GETL (BF) BRING IN LINE OF CODE
GETE (IDIR, I4) FIND END OF LINE
2
B
SAI 0467


1
II
SET SOURCE CODE INDICATOR SCIND = ER(13)
2,3
C
SET FORWARD/ BACKWARD TO BACK FBR = HB
IC (HNOD, BF) INITIAL ENTRY TO NODE TABLE
SET FORWARD/ BACKWARD FLAG TO FORWARD FBR = HF
END OF TABLE (IF.EQ.2)
YES
SET FORWARD/ BACKWARD TO BACK FBR = HB
1
A
NO
GETE (HNOD, I1)
SET STATEMENT TYPE STATYP = ER(3)
3
D
ASSIGN STATEMENT STATYP.EQ.38
YES
GETE (HNOD, I7) GET FIRST STATEMENT NUMBER
NO
FSTAT = ER(13) SET TO VARIABLE TO NO.
PAUSE STATEMENT STATYP.EQ.50
YES
GETE (HNOD, I8) GET LAST STATEMENT NO.
STATYP EQ 45 STATEMENT A GOTO
YES
3
F
SET I/O VARIABLE TO LAST STATEMENT NO. LSTAT = ER(13)
NO
2
C
1001 FORMAT (5(2x, I5), 2x, 'ERR230. .', 3(2x, I5), 2A4) PARAMETER LIST: I1, SCIND, FSTAT, LSTAT, K, IO, IO, IO, IO
WRITE ERROR TO FLAGFL
2
C
SAI 0468

UNIT MODULE DESCRIPTION

IDENTIFICATION

ER290

STORAGE ALLOCATION REQUIREMENTS (estimate)

4K hexadecimal bytes

PURPOSE

Flag COMMON variables multiply defined in DATA statements.

DESCRIPTION

This routine examines all COMMON blocks. Each module containing that COMMON block is examined for DATA statements. At this point the parameter in the COMMON block is compared against the parameters in the DATA statements. A record is kept only for each module of multiple definitions. Multiple definitions will be written on the FLAG FILE.

HOW ENTERED

AIR subroutine calls this subroutine when query 290 is requested.

CALLING SEQUENCE

CALL ER290

ROUTINES CALLED

CONALC	IE
CONALP	POP
EQUIVL	PUSH
GETE	TT

SET/USE PARAMETERS

SET

GLOBAL: COMMON/SPEREG/

FBR -- program flow flag

COMMON/ALINFO/

NAME(2) -- name save variable

MNAME(2,2) -- module name save variable

SCIND(2) -- source code index

FSTAT(2) -- first statement number of source code modules

LSTAT(2) -- last statement number of source code modules

NUMOCC -- error counter

USE

GLOBAL: COMMON/ALI/

ALIGN (2, 300) -- alignment table

PLALI(2) -- alignment table indicators

COMMON/FLAG

FLAGFL -- I/O variable for error information

COMMON/H/

HB -- hollerith B

HF -- hollerith F

COMMON/LTS/

LISTAB(500) -- list of equivalenced variables

LLIS -- list table pointers

PLLIS -- list table pointers

MAP(6,20) -- list table map

PMAP -- list table map pointer

PLMAP -- list table map pointer

COMMON/SPEREG/

ER(10) -- table information storage

USE

COMMON/TABLE/

HCOM -- hollerith COM for Common block table
HDIR -- hollerith DIR for Directory
HMAP -- hollerith MAP for Map table
HNOD -- hollerith NOD for Node table
HSYM -- hollerith SYM Symbol table
HUSE1 -- hollerith USE1 Use table
HUSE2 -- hollerith USE2 for table

SIGNIFICANT INTERNAL VARIABLES

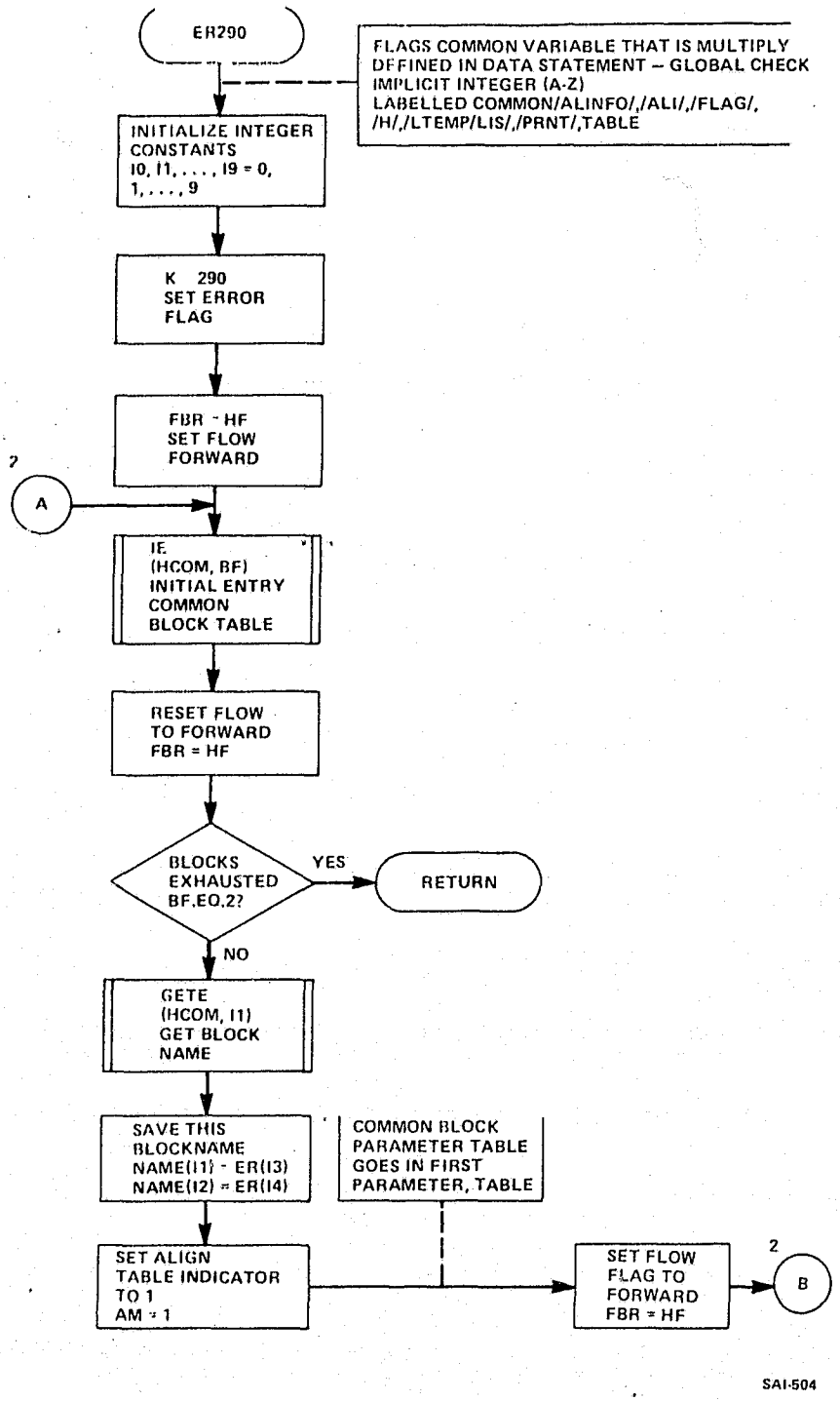
AM -- Indicates which alignment table to be filled
OVFLAG -- overflow flag for tables
ERFLAG -- irretrieveable error flag
ERINC -- counts 290 error occurrences

LIMITATIONS AND RESTRICTIONS

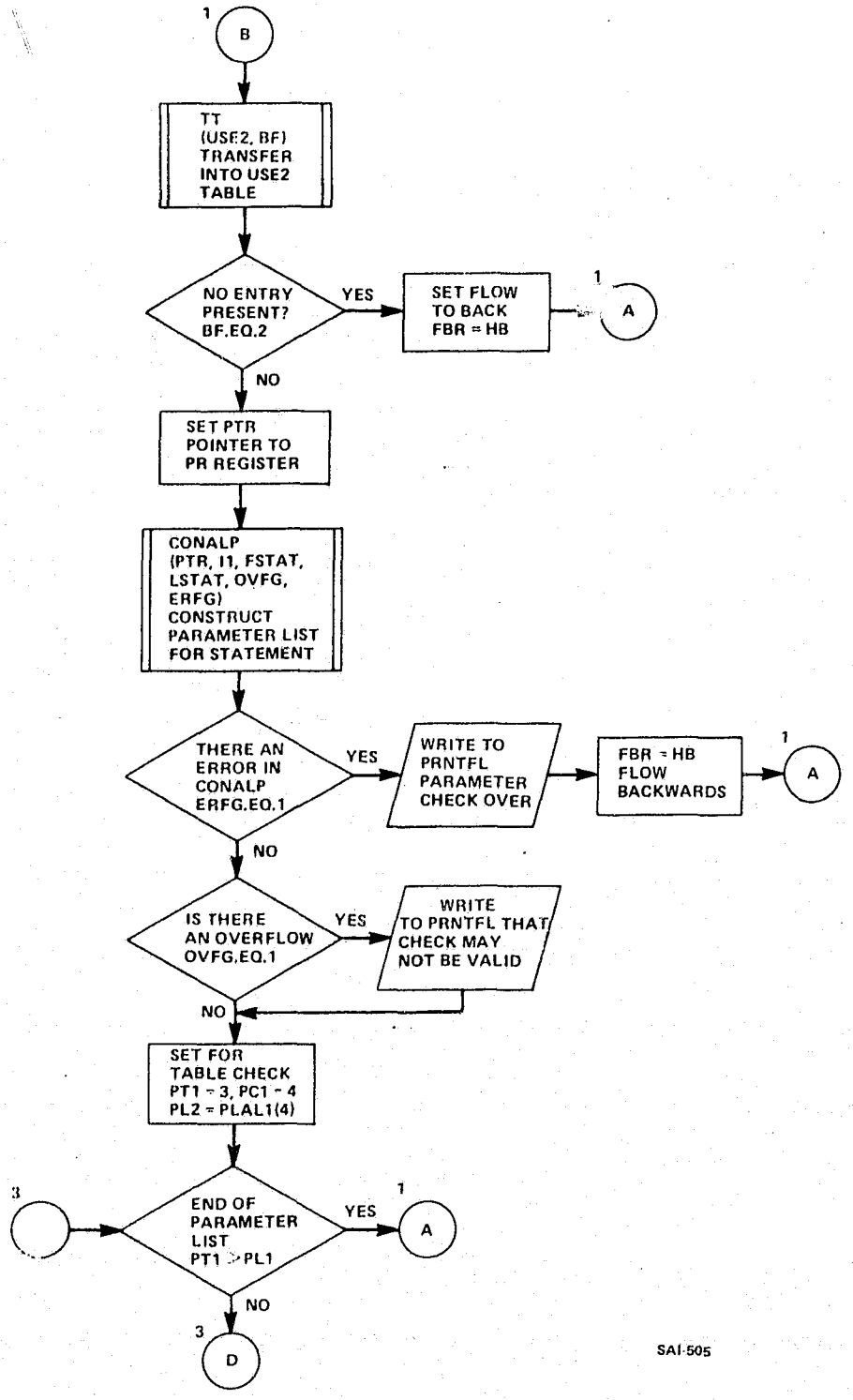
All variables are set with IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

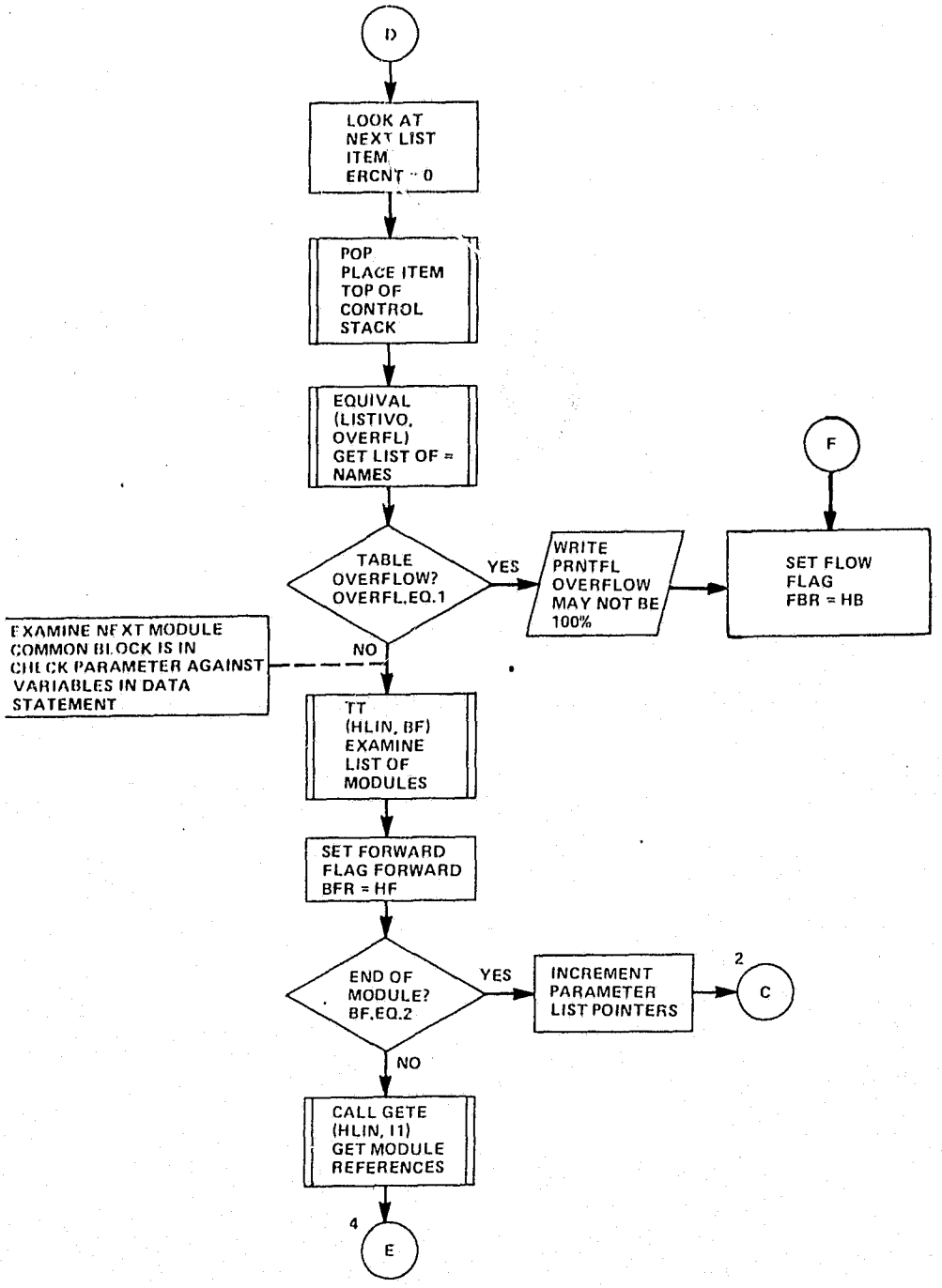
Attached

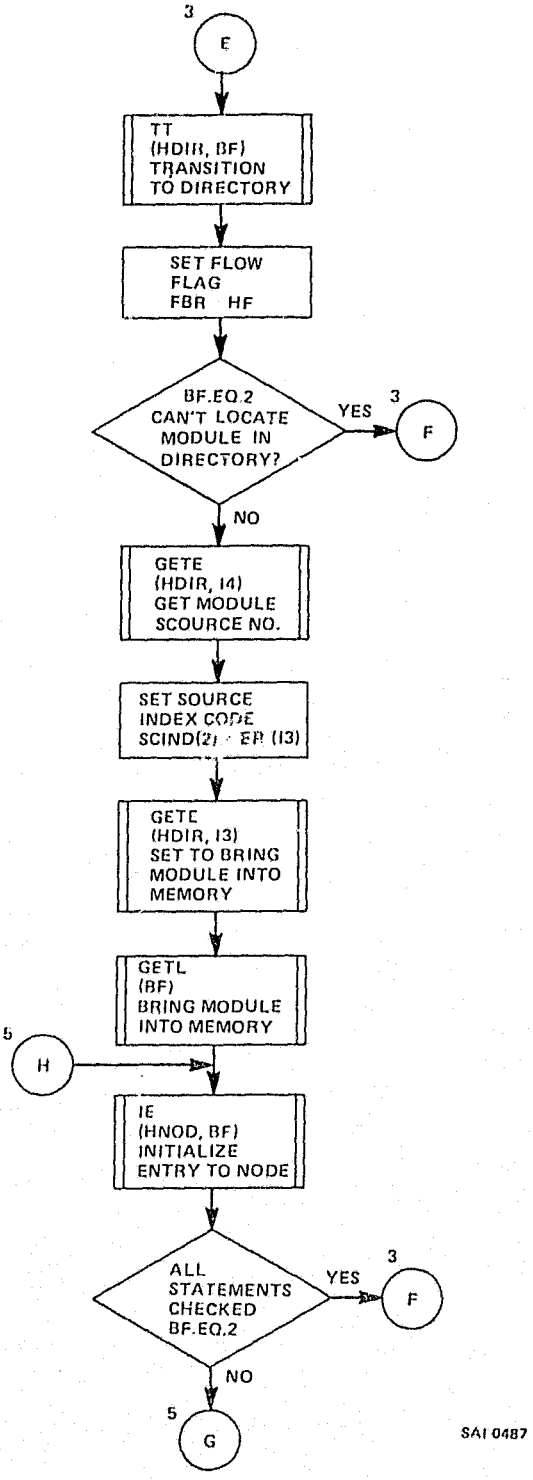

ER290
FLAGS COMMON VARIABLE THAT IS MULTIPLY DEFINED IN DATA STATEMENT – GLOBAL CHECK
IMPLICIT INTEGER (A-Z)
LABELLED COMMON/ALINFO/,/ALI/,/FLAG/,
/H/,/LTEMP/LIS/,/PRNT/,TABLE
INITIALIZE INTEGER CONSTANTS I0, I1, . . . , I9 = 0, 1, . . . , 9
K 290
SET ERROR FLAG
FBR = HF
SET FLOW FORWARD
2
A
IE
(HCOM, BF)
INITIAL ENTRY COMMON BLOCK TABLE
RESET FLOW TO FORWARD
FBR = HF
BLOCKS EXHAUSTED BF.EQ.2?
YES
RETURN
NO
GETE
(HCOM, I1)
GET BLOCK NAME
SAVE THIS BLOCKNAME
NAME(I1) = ER(I3)
NAME(I2) = ER(I4)
COMMON BLOCK PARAMETER TABLE GOES IN FIRST PARAMETER, TABLE
SET ALIGN TABLE INDICATOR TO 1
AM = 1
SET FLOW FLAG TO FORWARD
FBR = HF
2
B

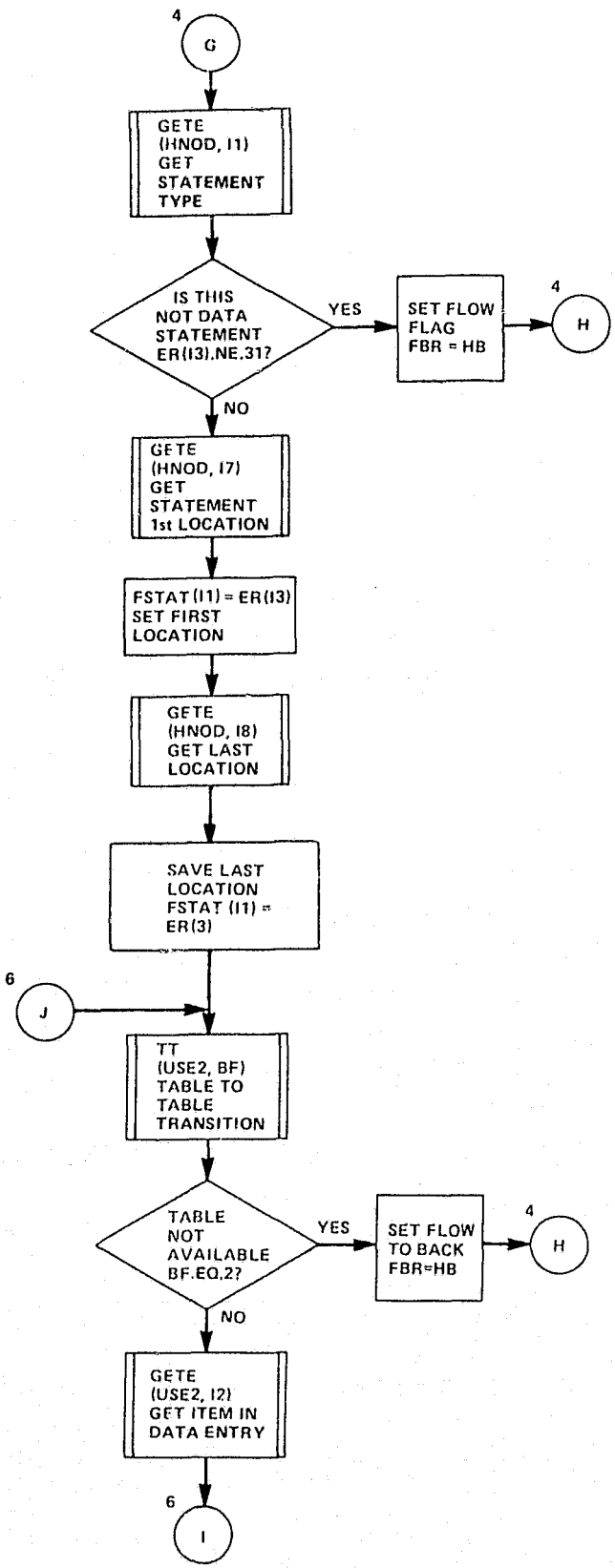
SAI-504



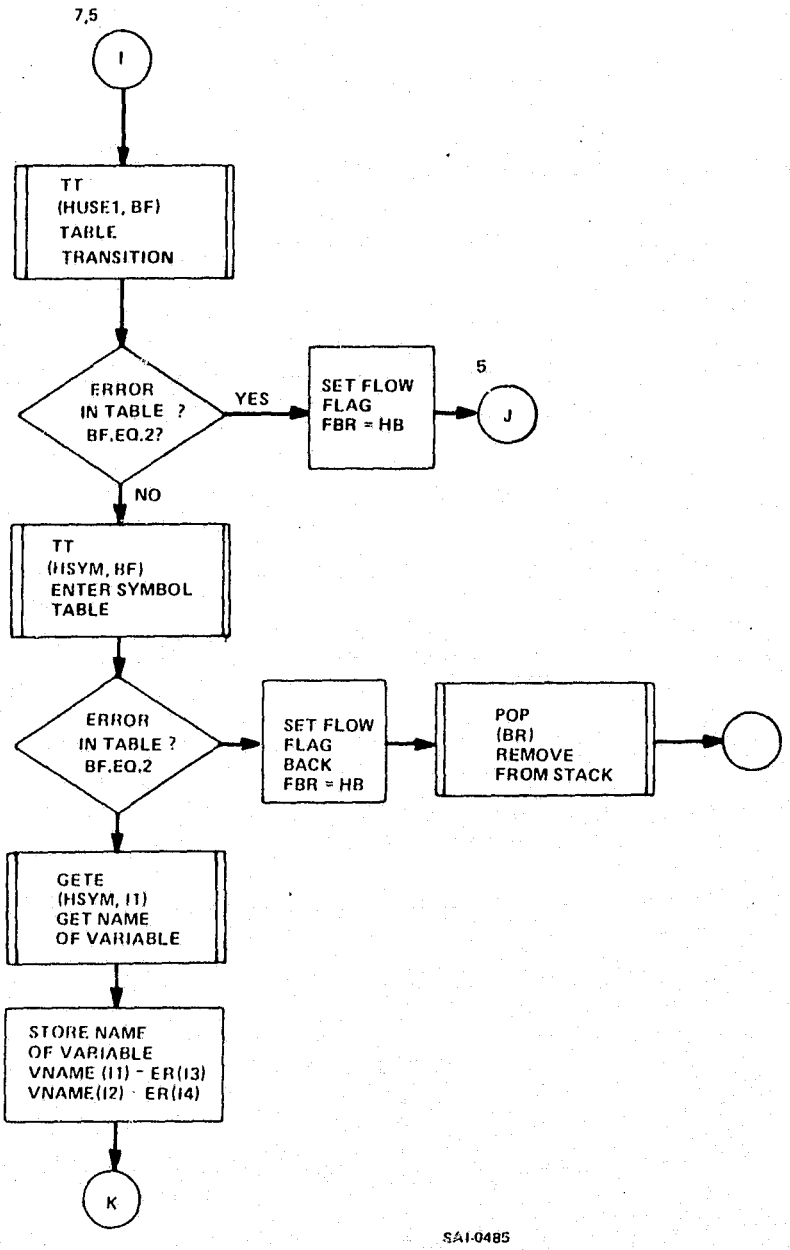
SAI-505


D
LOOK AT NEXT LIST ITEM ERCNT = 0
POP PLACE ITEM TOP OF CONTROL STACK
EQUIVAL (LISTIVO, OVERFL) GET LIST OF = NAMES
TABLE OVERFLOW? OVERFL.EQ.1
YES
WRITE PRNTFL OVERFLOW MAY NOT BE 100%
F
SET FLOW FLAG FBR = HB
NO
EXAMINE NEXT MODULE COMMON BLOCK IS IN CHECK PARAMETER AGAINST VARIABLES IN DATA STATEMENT
TT (HLIN, BF) EXAMINE LIST OF MODULES
SET FORWARD FLAG FORWARD BFR = HF
END OF MODULE? BF.EQ.2
YES
INCREMENT PARAMETER LIST POINTERS
2
C
NO
CALL GETE (HLIN, I1) GET MODULE REFERENCES
4
E

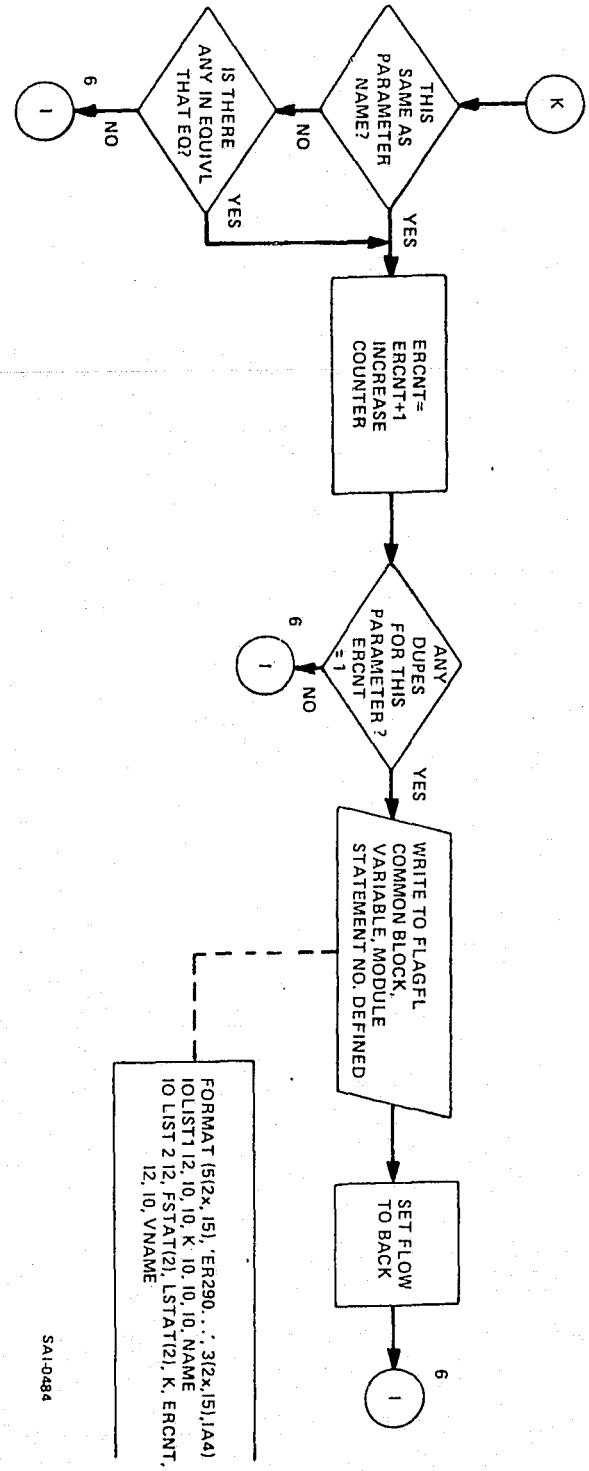

3
E
TT
(HDIR, BF)
TRANSITION
TO DIRECTORY
SET FLOW
FLAG
FBR HF
BF.EQ.2
CAN'T LOCATE
MODULE IN
DIRECTORY?
YES
3
F
NO
GETE
(HDIR, I4)
GET MODULE
SCOURCE NO.
SET SOURCE
INDEX CODE
SCIND(2) = ER (I3)
GETE
(HDIR, I3)
SET TO BRING
MODULE INTO
MEMORY
GETL
(BF)
BRING MODULE
INTO MEMORY
5
H
IE
(HNOD, BF)
INITIALIZE
ENTRY TO NODE
ALL
STATEMENTS
CHECKED
BF.EQ.2
YES
3
F
NO
5
G
SAI 0487



SAI-0490


7,5
I
TT
(HUSE1, BF)
TABLE
TRANSITION
ERROR
IN TABLE ?
BF.EQ.2?
YES
SET FLOW
FLAG
FBR = HB
5
J
NO
TT
(HSYM, BF)
ENTER SYMBOL
TABLE
ERROR
IN TABLE ?
BF.EQ.2
SET FLOW
FLAG
BACK
FBR = HB
POP
(BR)
REMOVE
FROM STACK
GETE
(HSYM, I1)
GET NAME
OF VARIABLE
STORE NAME
OF VARIABLE
VNAME (I1) - ER(I3)
VNAME(I2) - ER(I4)
K

SAI-0485


K
THIS SAME AS PARAMETER NAME?
NO
YES
IS THERE ANY IN EQUIVL THAT EQ?
NO
YES
6
1
ERCNT= ERCNT+1 INCREASE COUNTER
ANY DUPES FOR THIS PARAMETER ? ERCNT = 1
NO
YES
6
1
WRITE TO FLAGFL COMMON BLOCK, VARIABLE, MODULE STATEMENT NO. DEFINED
FORMAT (5(2x, I5), 'ER290. . .', 3(2x,I5),1A4)
IOLIST1 I2, I0, I0, K, I0, I0, I0, NAME
IO LIST 2 I2, FSTAT(2), LSTAT(2), K, ERCNT,
I2, I0, VNAME
SET FLOW TO BACK
6
1
SAI-0484

5. REFERENCES

1. Frank DeRemer and Hans Kron, "Programming-In-The-Large Versus Programming-In-The-Small," Proceedings 1975 International Conference on Reliable Software, 114-121 (1975).

2. B. H. Liskov, "A Design Methodology for Reliable Software Systems," Proceedings Fall Joint Computer Conference, 191-199 (1972).

3. Larry L. Constantine, "Structure Charts, A Guide," unpublished manuscript (1975).

4. Kathleen Jensen and Niklaus Wirth, PASCAL User Manual and Report, Springer-Verlag, New York, N.Y. (1975).

5. Peter Naur (ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM 6, 1-17 (Jan. 1963).

6. D.I. Good and L. C. Ragland, "NUCLEUS - A Language of Provable Programs," In William C. Hetzel (ed.), Program Test Methods, Prentice-Hall, Englewood Cliffs, N.J., 29-40 (1973).

7. E. Lohse (ed.), "Correspondence of 8-Bit Hollerith Codes for Computer Environments," Comm. ACM 11, 783-789 (Nov. 1968).

8. P. Henderson and R. Snowdon, "An Experiment in Structured Programming," BIT 21, 38-53 (1972).